Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation

A Dissertation submitted to

Fachbereich 18
Darmstadt University of Technology
Germany

Université Joseph Fourier
Grenoble I
France

to obtain the bi-national degree (Co-tutelle de thèse)* of
Doctor in Electrical Engineering

Gerd RITTER

Jury

President:	Prof. H. L. Hartnagel
Thesis Supervisors:	Prof. D. Borrione
	Prof. H. Eveking
Jury Members:	Prof. P. Bakowski
	Prof. M. Glesner

Day of submission:	21.12.2000
Day of defense:	26.03.2001

Research performed at

Dept. of Electrical and
Computer Engineering
Darmstadt University of Technology

TIMA Laboratory
Université Joseph Fourier
Grenoble

* arrêté ministériel du 5 Juillet 1984, du 23 Novembre 1988 et du 18 Janvier 1994

Contents

Abstract

A new approach to sequential verification of designs at different levels of abstraction by symbolic simulation is proposed. The automatic formal verification tool has been used for equivalence checking of structural descriptions at rt-level and their corresponding behavioral specifications. Gate-level results of a commercial synthesis tool have been compared to specifications at behavioral or structural rt-level. The specification need not be synthesizable nor cycle equivalent to the implementation. In addition, a future application of the method to property verification is proposed.

Symbolic simulation is guided along logically consistent paths in the two descriptions to be compared. An open library of different equivalence detection techniques is used in order to find a good compromise between accuracy and speed. Decision diagram (OBDD) based techniques detect corner-cases of equivalence. Graph explosion is avoided by using the results of the other equivalence detection techniques and by representing only small parts of the verification problem by decision diagrams. The cooperation of all techniques as well as good debugging support are made feasible by notifying detected relationships at equivalence classes instead of manipulating symbolic terms.

Keywords:

formal verification, symbolic simulation, equivalence checking, sequential verification, hardware verification, gate-level, rt-level

Kurzfassung

Ein neuer Ansatz zur sequentiellen Verifikation von Entwürfen auf verschiedenen Abstraktionsebenen durch symbolische Simulation wird vorgestellt. Das automatische formale Verifikationswerkzeug wurde dazu verwendet, die Äquivalenz von strukturellen Beschreibungen auf Registertransferebene und den entsprechenden Verhaltensspezifikationen nachzuweisen. Die Ergebnisse eines kommerziellen Synthesewerkzeugs auf Gatterebene konnten mit Verhaltens- bzw. Strukturbeschreibungen auf Registertransferebene verglichen werden. Es ist nicht erforderlich, daß die Spezifikation synthetisierbar oder taktäquivalent zur Implementierung ist. Ferner wird eine Anwendungsmöglichkeit der Methode zur Eigenschaftsverifikation vorgeschlagen.

Die symbolische Simulation wird entlang logisch konsistenter Pfade in den Beschreibungen durchgeführt. Eine erweiterbare Bibliothek verschiedener Techniken zur Äquivalenzerkennung erlaubt es, einen günstigen Kompromiß zwischen Genauigkeit und Geschwindigkeit zu erzielen. Auf Entscheidungsdiagrammen (OBDD) basierende Methoden erkennen seltene Fälle der Äquivalenz symbolischer Terme. Durch Einbeziehung der Resultate der anderen Techniken zur Äquivalenzerkennung gelingt es, die Größe der Graphen zu kontrollieren. Außerdem bilden die Entscheidungsdiagramme lediglich kleine Ausschnitte des Verifikationsproblems ab. Die Kooperation aller Techniken und eine effiziente Unterstützung der Fehleranalyse werden ermöglicht, indem Erkenntnisse über Termbeziehungen an Äquivalenzklassen vermerkt werden, anstatt die symbolischen Terme selbst zu manipulieren.

Schlüsselwörter:

formale Verifikation, symbolische Simulation, Äquivalenzprüfung, sequentielle Verifikation, Hardwareverifikation, Gatterebene, Registertransferebene

Résumé

Nous proposons une nouvelle méthodologie de simulation symbolique, permettant la vérification des circuits séquentiels décrits à des niveaux d'abstraction différents. Nous avons utilisé un outil automatique de vérification formelle afin de montrer l'équivalence entre une description structurelle précisant les détails de réalisation et sa spécification comportementale. Des descriptions au niveau portes logiques issues d'un outil de synthèse commercial ont été comparées à des spécifications comportementales et structurelles au niveau transfert de registres. Cependant, il n'est pas nécessaire que la spécification soit synthétisable ni qu'elle soit équivalente à la réalisation à chaque cycle d'horloge. Ultérieurement cette méthode pourra aussi s'appliquer à la vérification des propriétés.

La simulation symbolique est exécutée en suivant des chemins dont l'outil garantit la cohérence logique. Nous obtenons un bon compromis entre précision et vitesse en détectant des équivalences grâce à un ensemble extensible de techniques. Nous utilisons des diagrammes de décisions binaires (OBDD) pour détecter les équivalences dans certains cas particuliers. Nous évitons l'explosion combinatoire en utilisant les résultats des autres techniques de détection et en ne représentant qu'une petite partie du problème à vérifier par des diagrammes de décisions. La coopération de toutes les techniques, et la génération de traces permettant la correction d'erreurs, ont été rendues possibles par le fait que nous associons des relations à des classes d'équivalence, au lieu de manipuler des expressions symboliques.

Mots-clés:

vérification formelle, simulation symbolique, vérification d'équivalence, vérification séquentielle, vérification de matériel, niveau des portes logiques, niveau transfert de registres

List of Figures

List of Tables

Chapter 1

Introduction

Verifying the correctness of hardware designs is crucial in order to avoid substantial financial losses. Detecting a bug late in the design cycle can block important design resources and deteriorate the time-to-market. Validating a design with high-confidence and finding bugs as early as possible is therefore mandatory for chip design.

Numerical simulation with test-vectors is incomplete since only a non-exhaustive set of cases can be tested. It is also costly, as well in the simulation itself as in generating and checking the tests. Formal hardware verification covers all cases completely, and gives therefore a reliable positive confirmation if the design is correct.

The automatic formal verification technique described in this work combines *symbolic* simulation with a hierarchy of equivalence checking methods, including decision diagram based techniques. A complete verification of all cases is possible in contrast to numerical simulation since symbolic values are used. One symbolically simulated path corresponds in general to a large number of numerical simulation runs. During the symbolic simulation, relationships between symbolic terms are detected and recorded. A given verification goal like equivalence of the contents of relevant registers is checked at the end of each symbolic path.

Applications of formal verification techniques can be classified roughly in two types. *Property verification* checks whether a single design has some essential properties. *Equivalence checking* compares two descriptions of the same design and verifies whether a defined equivalence relation holds. The symbolic simulation technique has been successfully applied to equivalence checking of descriptions at different levels of abstraction. Therefore, the presentation of the approach in this document focuses on these verification problems, where experimental evidence exists. A possible future application to property verification is proposed.

The sequential behavior of two equivalent descriptions need not be identical. For example, significant modifications are often necessary to meet various requirements like costs, synthesizability, speed, timing constraints, power consumption etc. Equivalence often means that the specification and the implementation should produce the same result, but after a different number of control steps. Our symbolic simulation approach copes with such *sequential* verification problems, i.e., several control steps have to be considered to demonstrate the verification goal. An important advantage is the good debugging support of the automatic tool which can provide meaningful information about a counterexample to localize the design error.

Chapter 2 surveys the approach and presents the basic ideas. The application area and the scope of verification are described. Related work is discussed in chapter 3. Chapter 4 presents the implementation of the symbolic simulation approach in detail. Detecting the equivalence of symbolic terms is described separately since it represents the main part of the symbolic simulator. Chapter 5 presents the equivalence detection techniques used on the fly during the symbolic simulation. The more powerful, but less time-efficient equivalence detection based on decision diagrams is described in chapter 6. Experimental results and a conclusion are given in chapter 7 and 8.

Chapter 2

Overview of the Symbolic Simulation Approach

Section 2.1 discusses the essentials distinguishing our symbolic simulation approach from other methods. The verification scope is presented in section 2.2. Two examples, which cover only a small part of the application area, are used in section 2.3 to introduce the approach. Section 2.4 discusses how the values of registers being assigned in several cycles are distinguished. The representation of the descriptions for symbolic simulation is described in section 2.5.

Section 2.6 motivates why detecting equivalences of terms is the key for symbolic simulation. The use of *equivalence classes* during symbolic simulation is discussed. The principles of our hierarchical equivalence detection, which includes decision diagram based techniques, are given.

The presentation of the symbolic simulator in this work assumes for brevity that the verification goal is equivalence checking. Section 2.7 describes how other verification goals, in particular, property verification can be checked by the symbolic simulator, too. Finally, section 2.8 gives a short overview of the basic symbolic simulation algorithm.

2.1 Principles of Symbolic Simulation

The purpose of our verification approach is automatic sequential verification. Symbolic simulation is combined with a hierarchy of equivalence checking methods with increasing accuracy in order to optimize overall verification time without giving false negatives. Decision diagrams are flexibly used to detect corner-cases of equivalences. Only small parts of the verification problem are represented by decision diagrams to avoid graph explosion.

Sequential verification techniques relying on state space exploration cope with different abstraction levels but suffer from the state space explosion problem,

which limits their application area. Our symbolic simulation approach avoids state space traversal and copes also with memories.

Techniques denoted "symbolic simulation" or "symbolic evaluation" have been developed since the 1970s, chapter 3 gives some examples. The following essentials which are explained more detailed in the rest of the work distinguish our symbolic simulation approach, and permit a sequential verification at different levels of abstraction:

- symbolic terms are never manipulated, e.g., by canonizing or rewriting them; detected relationships, e.g., equivalence of terms are notified at equivalence classes instead;
- simulation is guided along valid, i.e., logically consistent paths in the descriptions instead of reducing the verification problem to a single formula which is checked afterwards;
- in most of the cases, only the information in the equivalence classes of the direct arguments is used to reveal equivalence between terms, i.e., tracing the expression trees of the arguments is avoided to permit a fast simulation;
- several register assignments along a valid path are explicitly distinguished instead of rewriting the register with the expressions assigned to it; therefore, term-size explosion is avoided.

Our contribution avoids a number of well-known deficiencies of other techniques which are discussed in chapter 3:

- theorem proving techniques require significant user interaction for our verification problems although they have a larger application area using general algorithms; our verification is automatic;
- techniques depending on state space exploration are not able to cope with the large state spaces of our examples;
- several techniques generate first a single huge formula to be checked afterwards; the formulas resulting especially from sequential verification at structural rt- or gate-level are often too complex for formula checkers; constructing a corresponding decision diagram for the verification problem leads to graph explosion; our techniques use decision diagrams, too, but only to check efficiently small parts of the problem.

A practically important advantage of the symbolic simulator is its good debugging support. Meaningful information about a counterexample or the successful verification can be provided. Verification is independent of the synthesis tools used, and copes with manual modifications by the designer.

2.2 Verification Scope

The symbolic simulator performs *automatic interpreted sequential* verification:

- *automatic*: the user needs no insight into the verification process;
- *interpreted*: demonstrating the verification goal requires an interpretation of functions;
- *sequential*: our symbolic simulator performs not only logic verification or combinational equivalence checking; sequential verification involves several control steps or cycles to demonstrate the verification goal.

The descriptions to be verified have to be *acyclic*. Loops need to be replicated according to the *maximum* number of executions.[1] For many cyclic designs with infinite loops the verification problem can also be reduced to an equivalence check of acyclic sequences, which is described in section 4.1.3.

Chapter 7 reports experimental results for the verification of the *computational equivalence* of two designs. Two descriptions are computationally equivalent if both produce the same final values on the same initial values; a formal definition is given in section 2.7. However, the scope of the symbolic simulation approach is larger than equivalence checking. Section 2.7 describes how other verification goals, particularly concerning *property verification*, can be demonstrated by performing an equivalence check.

Symbolic simulation can be used to verify the computational equivalence of descriptions at different levels of abstraction. Fig. 2.1 summarizes graphically the scope of the simulator:

- *rtl against rtl*: the descriptions can have different implementation details and the number of control steps to compute a result may vary;
  - *behavioral-rtl against behavioral-rtl*: experimental results for the verification of automatically constructed pipelined processors were presented first in [HER99]. The results in [RHE99] demonstrate that our symbolic simulation also copes with distinct orders of memory operations in the two descriptions to be compared;
  - *behavioral-rtl against structural-rtl*: the structural implementation of an architecture with microprogram control has been compared to behavioral specifications in [REH99]. The implementation details of the structural description and the fact that a different number of sequential steps has to be considered makes verification complex. Verification results for structural descriptions with different implementation details of pipelined DLX-processors are reported in [REH99], too;

[1] An empty loop body is simulated if the number of executions is smaller.

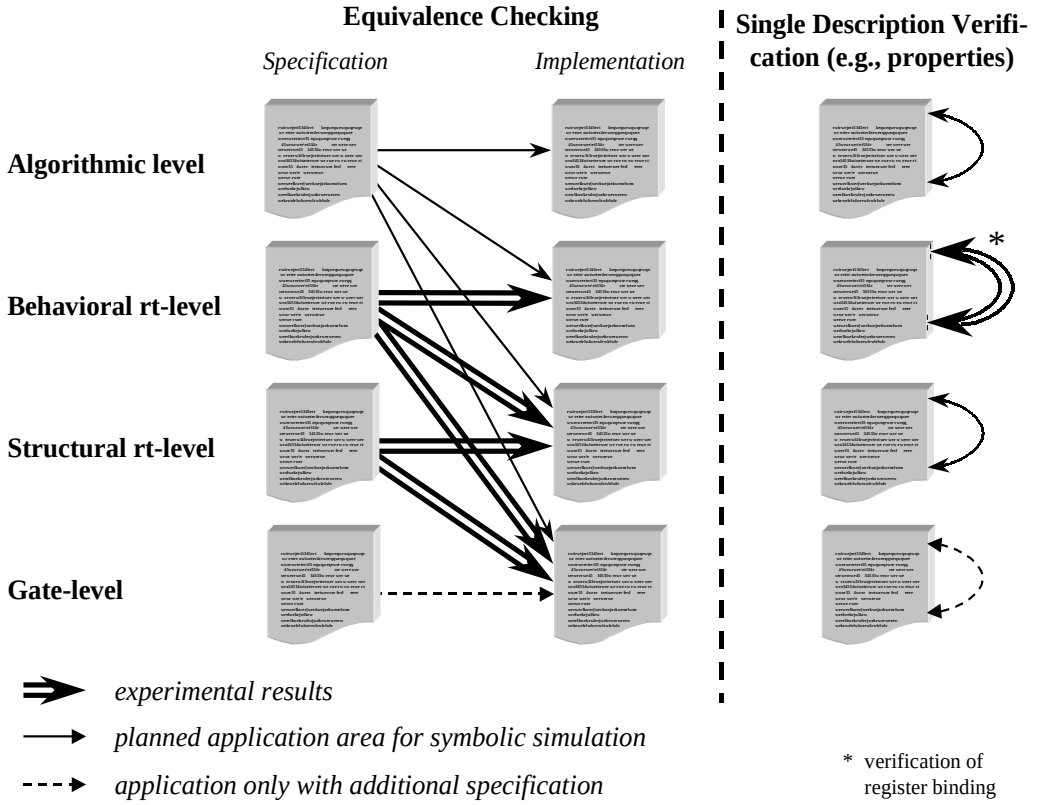


Fig. 2.1: Scope of the symbolic simulation approach

- *rtl against gate-level*: symbolic simulation copes not only with logic verification of cycle-equivalent descriptions but can also be used if several control steps have to be considered to demonstrate computational equivalence of the descriptions. The application to gate-level verification was described first in [Rit00];

- *algorithmic-level against rt-, algorithmic-, or gate-level* is a current research topic. A compiler which translates a subset of ANSI C into the experimental language of the simulator is described in [Lev00]. Verification is limited by loops which have to be unrolled as described in section 4.1.3;

- *single description verification*: a first application to verification of register binding was presented in [BRHE00, Bla00], see also section 7.4.

Fig. 2.1 indicates that a symbolic simulation of a single description *at gate-level*, e.g., for property verification can be problematic. No case-splits are performed during the simulation of the gate-level description. Therefore, the entire verification task is concentrated on a single symbolic simulation run, which makes equivalence detection difficult. The same holds if two descriptions at gate-level

are compared, see left-hand side of Fig. 2.1 (dotted arrow).[2] Providing a specification at a higher abstraction level allows also verifying these gate-level problems. The simulation of the specification at higher level is used to "guide" the path search or symbolic simulation of the gate-level description, see section 4.6. The verification task is divided since the specification defines the respective path to be simulated at gate-level.

2.3 Introductory Examples

Two examples are used to introduce the symbolic simulation approach. Note that these examples do *not* cover the verification scope as described in the previous section:

- the application area of the symbolic simulator to verify descriptions at different levels of abstraction is larger, see above,
- only equivalence checking is considered, and
- a sequential verification over several cycles is necessary for both examples, but the intermediate results are the same; this is not required for computational equivalence.

The first example (rt-level⇔rt-level) is used to give a first idea of the basic simulation procedure. The second example introduces verification at gate-level. Section 4.5 describes the symbolic simulation of both examples by the implemented verification tool.

Example 2.1
Fig. 2.2 describes two computationally equivalent parts of two descriptions at rt-level. Equivalence is given with respect to the final value of the register `r`.

The equivalence checker simulates symbolically all possible paths. False paths are avoided by making only consistent decisions at branches in the description. A case-split is performed if a condition is reached which cannot be decided but depends on the initial register and memory values, e.g., `opcode(m)=101` in Example 2.1. The example requires the symbolic simulation of two paths since the other condition `z=101` has to be decided consistently. Note that both symbolic paths represent an important number of "classical" simulation runs.

Each symbolically executed assignment establishes an equivalence between the destination variable on the left and the term on the right side of an assignment.

[2] This verification step can be done efficiently by other techniques, e.g., combinational equivalence checking if the circuit is not retimed.

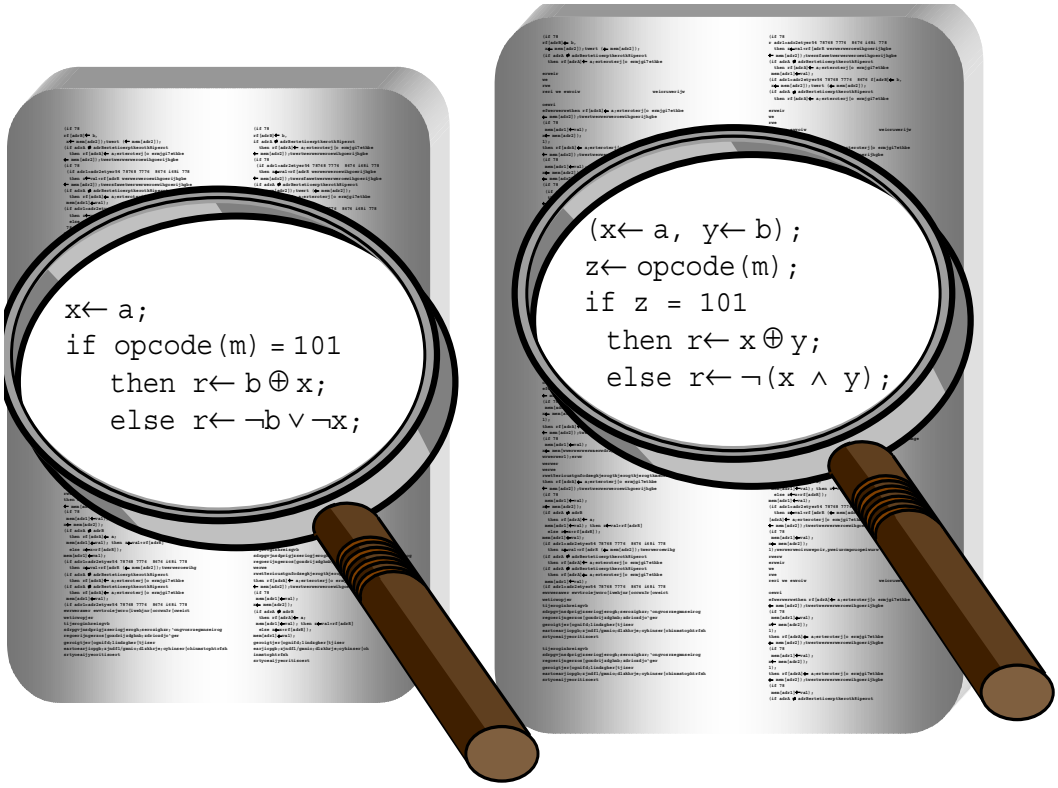


Fig. 2.2: Example for rtl ⇔ rtl verification

Additional equivalences between terms are detected during simulation. Equivalent terms are collected in equivalence classes. During the path search, only relationships between terms that are fast to detect or that are often crucial to check the verification goal are considered on the fly. Some functions remain uninterpreted while others are more or less interpreted to detect equivalences of terms, which are considered by unifying the corresponding equivalence classes.

Having reached the end of both descriptions with consistent decisions, a complete path is found and the verification goal is checked for this path, e.g., if both produce the same final values of `r`. This check is trivial for the `then`-branches in Fig. 2.2 since the equivalence of $\mathtt{b} \oplus \mathtt{x}$ and $\mathtt{x} \oplus \mathtt{y}$ is detected on the fly.

Using only a selection of function properties for equivalence detection during the path search which are fast to compute, we may fail to prove the equivalence of two terms at the end of a path, e.g., the equivalence of $\neg\mathtt{b} \vee \neg\mathtt{x}$ and $\neg(\mathtt{x} \wedge \mathtt{y})$ in the `else`-branches of Fig. 2.2. The application of De Morgan's Law on *bit-vectors* in this example is not detected during symbolic simulation. In these cases the equivalence of the final values of `r` is checked using *decision diagrams*. If this fails, it is verified whether a false path is reached, since conditions may be decided inconsistently during the path search due to the limited equivalence detection. If the decisions are sound, the counterexample for debugging is reported. Relevant details about the symbolic simulation run can be provided since all information is available on every path.

Example 2.2

Fig. 2.3 compares a specification at rt-level and an implementation at gate-level.

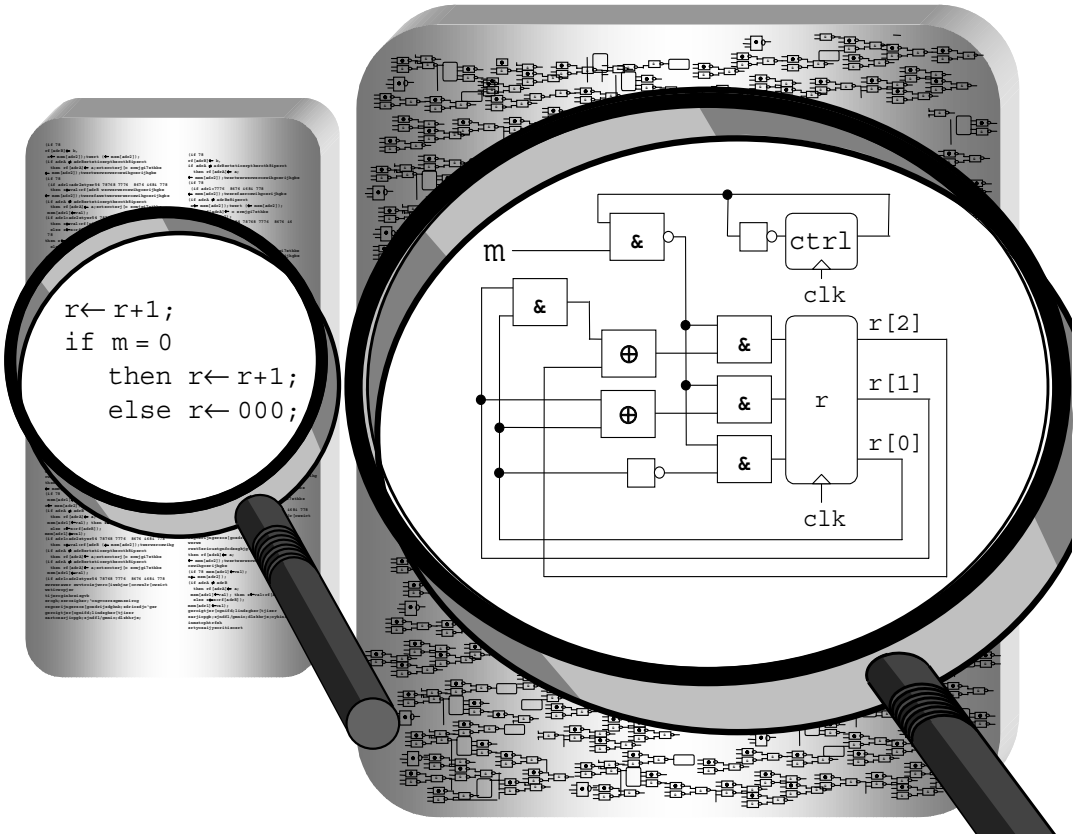


Fig. 2.3: Example for rtl ⇔ gate-level verification

They are computational equivalent with respect to the register `r` *if* `ctrl` *is initialized with 0 and if the execution takes two cycles. The implementation at gate-level includes the signal assignments to the three bits of the register* `r` *and to the control flag* `ctrl`*. Two cycles of symbolic simulation are required to demonstrate equivalence. In the first cycle,* `r+1` *is calculated and* `ctrl` *is set true. The if-then-else-clause evaluating the flag* `m` *is considered in the next cycle. Symbolic simulation has to demonstrate that the final values of* `r` *are the same.*

Two cycles have to be simulated symbolically in the example of Fig. 2.3. Therefore, the gate-level description representing only one cycle is put together for two times before simulating, i.e., the description is replicated accordingly to the number of cycles required. The values of the registers of the previous simulation cycle are the input values of the next cycle, see Fig. 2.4.

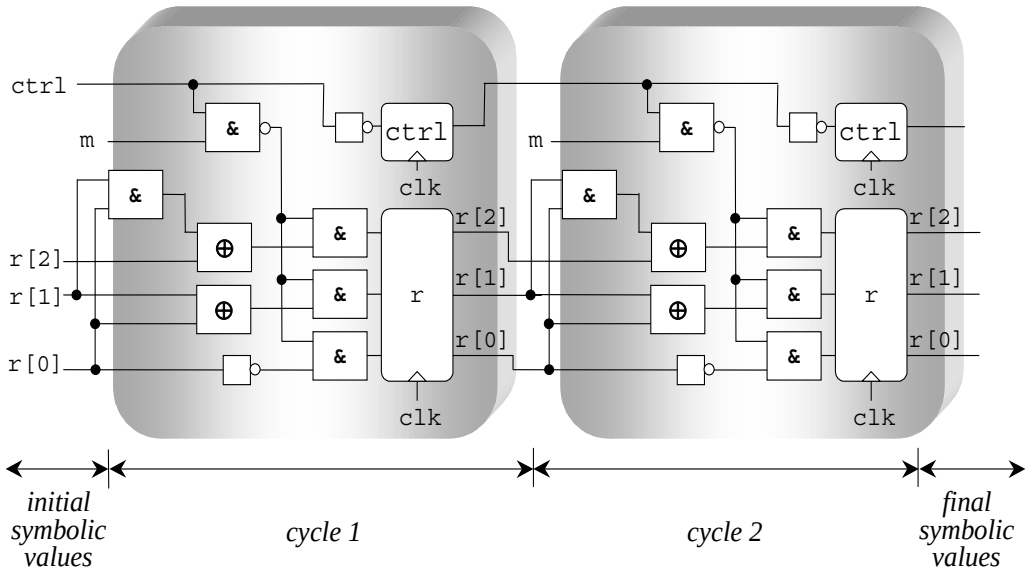


Fig. 2.4: Duplicating a gate-level description

2.4 Distinguishing Different Register Values

The values of each register being assigned in several cycles are distinguished by indexing. We do not substitute the register in the following by the symbolic term assigned to it to avoid term-size explosion. An indexed register name is called a *RegVal*. A new *RegVal* with an incremented index is introduced after each assignment to a register. An additional upper index s or i distinguishes the *RegVals* of the specification and of the implementation. For example, `ar←a+b;` is replaced by $\texttt{ar}_2^s \leftarrow \texttt{a}_1^s\texttt{+b}_1^s\texttt{;}$ in the specification if all registers have already been assigned once. Only the initial $\mathit{RegVals}_{initial}$ as anchors are identical in the specification and in the implementation, since the equivalence of the two descriptions is tested with respect to arbitrary but identical initial register values.

RegVals are also used to distinguish the different states of a memory. A new *RegVal* with an incremented index is introduced after each `store`-operation to a memory. For example, the third `store`-operation to a memory `mem[adr]←val;` becomes $\texttt{mem}_3^s \leftarrow \texttt{store}(\texttt{mem}_2^s, \texttt{adr}_1^s, \texttt{val}_1^s)$. The *RegVals* $\texttt{mem}_2^s$ and $\texttt{mem}_3^s$ represent the memory state before and after the `store`-operation.

Definition 2.1 (RegVal)
A RegVal represents

- *the initial symbolic value of a register,*
- *the symbolic value of a register after an assignment until the next assignment to the same register,*

- *the initial symbolic state of a memory, or*
- *the symbolic state of a memory after a* `store`*-operation until the next* `store`*-operation to the same memory.*

2.5 Internal Representation for Symbolic Simulation

The descriptions simulated symbolically consist of:

- lists of assignments to *RegVals*; the expressions assigned are other *RegVals*, constants, or terms, i.e., functions of *RegVals*; note that memory access is modeled by `read`- and `store`-operations;
- *if-then-else*-clauses; both branches can contain a list of assignments to *RegVals* and/or several *if-then-else*-clauses. Symbolic simulation forks at each *if-then-else*, which requires a case-split on the corresponding condition.

Parallel assignments are considered implicitly by the indexes of the *RegVals*. Other control structures, e.g., *case*-clauses or multiplexers are compiled into *if-then-else*-clauses.[3]

In general, at least one of the descriptions to be compared contains *if-then-else*-clauses. Gate-level descriptions consist only of assignments to *RegVals*. Intermediate signals are either substituted by the corresponding expression until primary inputs or the output of flip-flops is reached; or they are considered for technical reasons as "artificial" *RegVals*.[4] Primary inputs are modeled by *RegVals*, too. Compilation of descriptions at structural or behavioral rt-level is straightforward. Section 4.1 describes the preparation of the data structure.

2.6 Detecting Equivalences of Symbolic Terms

Symbolic simulation argues about symbolic terms which represent a set of different values. The actual value selected from this set depends on the *initialization* of the registers and memories. Deciding whether two terms are equivalent is trivial in numerical simulation, but not obvious if *symbolic* terms are used. Intuitively, two terms or *RegVals* are equivalent if an exhaustive numerical simulation of each possible initialization produce in all cases the same value for both terms.

[3] Note that this is only an implementation choice.

[4] The corresponding expression is assigned to this "artificial" *RegVal*, see "*simulation-cutpoints*" in appendix 9.3.

Equivalence of two terms can depend on the actual path followed during symbolic simulation.

Definition 2.2 (Path)
Let $\mathcal{C}$ be a set of conditions. A path consists of associating the value true or false to each condition in $\mathcal{C}$.

The decisions of a path guarantee that specification and implementation can be simulated until both ends are reached without requiring additional case-splits at *if-then-else*-clauses; i.e., no condition occurs which depends on the initial *RegVals* on the assumptions concerning $\mathcal{C}$. A *partial* path permits simulating without additional case-splits until the two ends of the partial path in both the specification and the implementation are reached. Note that a *branch* denotes only one of the two possibilities of a *single if-then-else*-clause, i.e., the `then`- or the `else`-branch.[5] A path comprises mostly decisions about conditions of more than one *if-then-else*-clause.

The following definition of equivalence considers complete and partial paths by referring to acceptable initializations. The set of combinations of acceptable initial *RegVals* is constrained:

- by the $domain(RegVal_{initial,k})$ of the *RegVal*; the index k distinguishes *RegVals* of different registers; the type of a register can be integer or bit-vector;[6] the bit-vector length of the register constrains the domain in the second case; additional restrictions can be defined by the user, i.e., to exclude impossible initializations;
- by case-splits, leading to one of the decisions about a condition; $\mathcal{C} = \{C_0, \cdots, C_n\}$ subsumes all conditions requiring a case-split.

Definition 2.3 (Evaluation of a term or *RegVal*)

$$eval(t) = \begin{cases} t \text{ is a constant} & : \quad t \\ t \text{ is a } RegVal_{initial,k} & : \quad init(RegVal_{initial,k}) \\ t \text{ is a } RegVal_{j \neq initial,k} & : \quad eval(t') \\ & \quad t' : \text{right-hand side term of} \\ & \qquad \text{assignment to } RegVal_{j,k} \\ t = F(a_0, \cdots, a_l) & : \quad F\left(eval(a_0), \cdots, eval(a_l)\right) \end{cases}$$

Definition of $eval(t)$ supposes that all registers and functions are typed with domains on which equality $=$ is available.

$eval(t)$ returns a constant for an acceptable initialization.

[5] `elsif`-clauses can be considered as sequences of *if-then-else*-clauses.

[6] Note that the decision diagram based tests described in chapter 6 are not applicable for integers if no information about the range is given.

Definition 2.4 (Acceptable initialization)
Acceptable initializations of the registers in the descriptions are:

$$
\begin{aligned}
&acceptable(init^{RegVals}) \Leftrightarrow \\
&\qquad (\forall RegVal_{initial,k} : init(RegVal_{initial,k}) \text{ is a constant} \wedge \\
&\qquad\qquad init(RegVal_{initial,k}) \in domain(RegVal_{initial,k})) \wedge \\
&\left(\begin{array}{lll} \forall C_i \in \mathcal{C} & : & eval(C_i) \text{ is a constant} \wedge \\ & & \left\{ \begin{array}{lll} C_i \text{ decided true} & : & eval(C_i) = 1 \\ C_i \text{ decided false} & : & eval(C_i) = 0 \end{array} \right. \end{array} \right)
\end{aligned}
$$

The evaluation of the conditions in $\mathcal{C}$ guarantees that a given initialization does not violate one of the decisions. The constants 1 and 0 represent true and false. Definition of an acceptable initialization supposes that any term used as condition in an if-then-else-clause evaluates to one of these values. An extension of the definition for RegVals of memories is given in section 4.1.5.

Definition 2.5 (Valid path)
A valid path - in contrast to a false path - implies that at least one acceptable initialization exists according to Definition 2.4.

Definition 2.6 (Equivalence of terms)
Two terms or RegVals t_1 and t_2 are term-equivalent $\equiv_{\mathcal{C}}$ if under the decisions taken previously on the path concerning the conditions $\mathcal{C} = \{C_0, \cdots, C_n\}$ their values are identical for any acceptable initialization of the RegVals:

$$t_1 \equiv_{\mathcal{C}} t_2 \Leftrightarrow \forall init^{RegVals} : acceptable(init^{RegVals}) \Rightarrow eval(t_1) = eval(t_2)$$

Equivalent terms are detected along valid paths, and collected in *equivalence classes* (*EqvClasses*). We write $term_1 \cong_{\mathcal{C}} term_2$ if two terms are in the same equivalence class established during simulation. If $term_1 \cong_{\mathcal{C}} term_2$ then $term_1 \equiv_{\mathcal{C}} term_2$. $\equiv_{\mathcal{C}}$ denotes that the two terms *are* equivalent according to Definition 2.6 while $\cong_{\mathcal{C}}$ means that the two terms have been *identified* during symbolic simulation to be $\equiv_{\mathcal{C}}$. Equivalence detection on the fly is incomplete as discussed below to permit a fast symbolic simulation. Therefore, the relationship $term_1 \equiv_{\mathcal{C}} term_2$ might be not revealed, i.e., the terms are still in different *EqvClasses*. The expression "equivalent" is used in the following as synonym for $term_1 \cong_{\mathcal{C}} term_2$.

Initially, each *RegVal* and each term gets its own equivalence class. Equivalence classes are unified in the following cases:

- two terms are identified to be equivalent by reasoning; the equivalence detection techniques used are presented in chapter 5 and 6;
- a condition is decided; if this condition is
  - a test for equality $a = b$, then the equivalence classes of both sides are unified *only if the condition is asserted*,

  - otherwise (e.g., $a < b$ or a status-flag) the equivalence class of the condition is unified with the equivalence class of the constant 1 or 0 if the condition is asserted or denied;

- after every assignment. Practically, this union-operation is significantly simpler because the equivalence class of the *RegVal* on the left-hand side of the assignment was not modified previously.

Equivalence classes permit to keep also track about *inequivalences* of terms:

Definition 2.7 (Inequivalence of terms)
Two terms or RegVals t_1 and t_2 are inequivalent $\not\equiv_{\mathcal{C}}$ if under the decisions taken previously on the path concerning the conditions $\mathcal{C} = \{C_0, \cdots, C_n\}$ their values are never identical for any acceptable initialization of the RegVals:

$$t_1 \not\equiv_{\mathcal{C}} t_2 \Leftrightarrow \forall init^{RegVals} : acceptable(init^{RegVals}) \Rightarrow eval(t_1) \neq eval(t_2)$$

Intuitively, two terms are inequivalent if an exhaustive numerical simulation of all possible initial register values and memory states produces in all cases different values for the two terms. We write $term_1 \not\cong_{\mathcal{C}} term_2$ or use the expression "inequivalent" if two terms are identified to be $\not\equiv_{\mathcal{C}}$ during simulation. Equivalence classes containing $\not\cong_{\mathcal{C}}$ terms are inequivalent, too. This is the case

- if different constants are members of the *EqvClasses*;
- if a condition with a test for equality (e.g., $a = b$) is decided to be false;
- if terms of the *EqvClasses* are identified to be inequivalent by reasoning.

Identifying inequivalences during symbolic simulation requires mostly no specialized techniques or is done using decision diagrams. The reason is that they are caused in most of the cases either by the fact that two terms are equivalent to different constants or by case-splits. On the other hand, detecting equivalences between symbolic terms is the most important task during symbolic simulation:

- equivalence is the strongest relationship which the two sets of possible values of two symbolic terms can have: the value for both symbolic terms is the same for any acceptable initialization of the registers and memories;
- conditions have to be decided consistently during symbolic simulation; conditions are often checks for equality, e.g., `a="0001"` which can be decided without case-split if the two terms are previously detected to be equivalent. All other conditions can also be considered as a check for equality to the constants 1 or 0, which represent the values *true* and *false*.

  Example 2.3
  The conditions `a<5`, `a[14]`, *or* `odd(a)` *are decided without case-split if the corresponding terms are equivalent to the constants 1 or 0;*

- knowledge about equivalence or inequivalence of two terms is the key information in most of the cases to decide the relationship of other terms;

  Example 2.4
  - `(not(x) and y)` *is equivalent to 0 if x is equivalent to y;*
  - `a+b` *and* `c+d` *are equivalent if the arguments are pairwise equivalent;*
  - *two* `read`*-operations from a memory result in the same value if the addresses are equivalent and no intervening* `store`*-operation exists;*
  - *if a two-bit vector is inequivalent to the constants 00, 01, and 10, then it is equivalent to the constant 11.*

  Chapter 5 discusses how knowledge about equivalences or inequivalences of the arguments can be used efficiently during symbolic simulation to discover relationships between terms by reasoning;

- verification goals other than equivalence checking, i.e., property verification can be reduced to a check for equivalence of terms, too, see section 2.7.

The techniques described in chapter 5 search equivalent terms on the fly depending on the function of this term. Ideally, all $\equiv_{\mathcal{C}}$ terms and *RegVals* are in the same equivalence class, but it is too time consuming to search for all possible equivalences on the fly. In order to speed up the path search, the following simplifications are made with respect to a complete equivalence detection:

- only fast to check or "crucial" properties of interpreted functions are considered;
- only the information of the equivalence classes of the direct arguments is used in most of the cases to reveal equivalences between terms; i.e., the equivalence of terms can be decided by simply testing if the arguments are $\cong_{\mathcal{C}}$ or $\ncong_{\mathcal{C}}$. Expanding the arguments, i.e., tracing the corresponding expression trees of the arguments is avoided to permit a fast simulation;
- invoking the equivalence detection techniques is restricted as described in section 4.2.

The incomplete equivalence detection on the fly permits a fast symbolic simulation but may fail to find the equivalence of two terms. Therefore, more accurate tests called *dd-checks* based on decision diagrams [Bry86] are used at the end of a path if the verification goal is not demonstrated. These more powerful, but also less time-efficient equivalence detection techniques using vectors of *OBDDs* are described in chapter 6.

Two terms are frequently equivalent or inequivalent only under the assumptions of previous case-splits constraining the set of possible initial *RegVals*, i.e., the relationship is *path-dependant*.

Example 2.5
A case-split is necessary in the specification of Fig. 2.5 since the value of a=b

Specification	**Implementation**
if (a=b) **then** $\cdots$	$\mathtt{x}_1^i \leftarrow$ a+b;
else $\cdots$	$\mathtt{y}_1^i \leftarrow$ b vand c;
$\mathtt{x}_1^s \leftarrow$ a+a;	
$\mathtt{y}_1^s \leftarrow$ a vand c;	

Fig. 2.5: Path-dependant equivalence/inequivalence

depends on the initial RegVals. The terms $\mathtt{x}_1^s$ *and* $\mathtt{x}_1^i$ *as well as* $\mathtt{y}_1^s$ *and* $\mathtt{y}_1^i$ *are equivalent in the case where* a=b *is asserted. The operator '*vand*' performs the bit-wise conjunction of the bit-vectors* a *and* c*. In the other case* $\mathtt{x}_1^s$ *and* $\mathtt{x}_1^i$ *are inequivalent since the additions result in different values no matter what the initialization under the assumption* $\mathtt{a} \not\cong_{\mathcal{C}} \mathtt{b}$*. The terms* $\mathtt{y}_1^s$ *and* $\mathtt{y}_1^i$ *are neither equivalent nor inequivalent since* c *might be initialized with zero. Therefore,* $\mathtt{y}_1^s$ *and* $\mathtt{y}_1^i$ *may or may not return the same values.*

Example 2.5 describes the three basic cases (ternary logic) which can be distinguished by using the information of the *EqvClasses*:

1. two terms are in the same *EqvClass*; the terms are $\cong_{\mathcal{C}}$;
2. two terms are in inequivalent *EqvClasses*; the terms are $\not\cong_{\mathcal{C}}$;
3. otherwise they either produce different values for *some* acceptable initialization of the *RegVals*, or equivalence/inequivalence has not yet been detected.

2.7 Rewriting Verification Goals

Checking computational equivalence in a given path consists simply of comparing the *EqvClasses* of the respective *RegVal*-pairs.

Definition 2.8 (Computational equivalence)
Two descriptions are computationally equivalent if both produce the same final values on the same initial values relative to a set of relevant variables. Let $\mathcal{C}$ *be as in Definition 2.6. For each path characterized by a number of case-splits leading to the decisions about the conditions in* $\mathcal{C}$*, the following relation must hold*

$$\forall paths, RegVals_k \in RegVals^{relevant} : RegVal^s_{final,k} \equiv_{\mathcal{C}} RegVal^i_{final,k}$$

$RegVal^{s/i}_{final}$ *are the corresponding RegVals in the specification and in the implementation with the highest index in the respective path.*[7]

[7]The number of assignments to a register can vary depending on the path. Therefore, the highest index might differ.

Note that not all final *RegVals* have to be equivalent for computational equivalence, i.e., there might be

- a subset of register/memories appearing only in the implementation, which can have arbitrary final values, e.g., additional pipeline-registers, and
- a subset of register/memories appearing in the specification which are not relevant for the equivalence check, e.g., the value of an instruction register.

The description of the symbolic simulation approach in the rest of this work refers to computational equivalence as verification goal. However, many other verification goals can be easily reduced to a check for computational equivalence or the simulation tool can easily be extended. For example, verifying if two descriptions are *trace-equivalent* [EHR99], i.e., if all runs coincide step-by-step, requires comparing not only the final *RegVals* but all pairs of intermediate *RegVals*. Note that one condition for *trace-equivalence* is that the number of sequential steps in the two descriptions has to be the same on all paths.

Property verification can often be reduced to a check for computational equivalence by introducing "fictive" registers which are used as control flags. Those flags are set on a path if the property is violated. If the annotated description is computationally equivalent to a "dummy"-specification which clears only the corresponding flag then the property is satisfied.

Example 2.6
The register binding verification described in section 7.4 requires checking if there is no path where a flag `check` *is set to 1 due to an incorrect register binding. The specification consists of an assignment* $\texttt{check}_1^s \leftarrow 0$*, see Fig. 2.6. The same assignment is added in front of the implementation. The constant 1 is assigned to* `check` *in the following, if a conflict of the register binding is discovered. The disjunction prevents resetting* `check` *in the following. The flag is never set, i.e., the register binding is correct iff the descriptions are computationally equivalent with respect to* `check`*.*

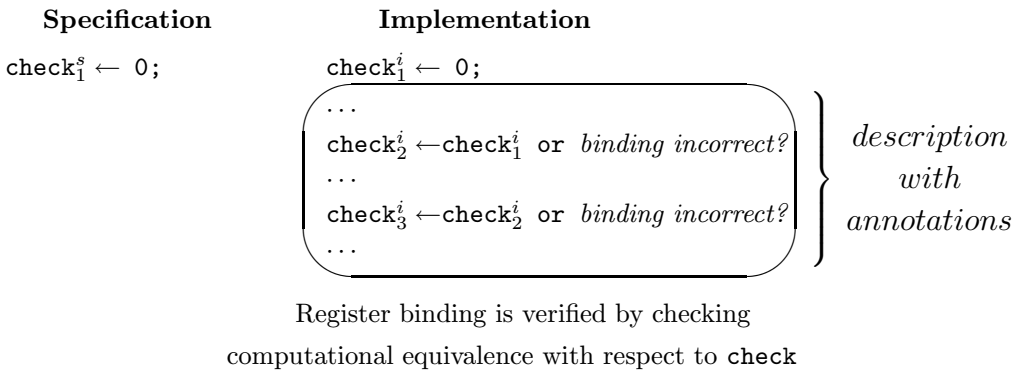


Fig. 2.6: Adding control flags for property verification

The verification of arbitrary properties is straightforward corresponding to Fig. 2.6. Usually, the condition *binding incorrect?* in Fig. 2.6 has to be replaced by the property to check.

Example 2.7
The following annotations are required to check if

- *bit 15 of a register* `reg` *is always cleared:* $\texttt{check}_n^i \leftarrow \texttt{check}_{n-1}^i$ `or` $\texttt{reg}_x^i$`[15];`
- *two arbitrary RegVals* $\texttt{r1}_x^i$ *and* $\texttt{r2}_y^i$ *are equivalent:* $\texttt{check}_n^i \leftarrow \texttt{check}_{n-1}^i$ `or not(`$\texttt{r1}_x^i \equiv \texttt{r2}_y^i$`);`
- *a register does not exceed the value 15:* $\texttt{check}_n^i \leftarrow \texttt{check}_{n-1}^i$ `or` $(\texttt{r}_x^i > 15)$.

Again, the symbolic simulator can provide meaningful information about the counterexample if the property is not satisfied. Note that inserting the annotations can be supported by the generation of the internal data structure described in section 4.1, 9.1, and 9.3. For example, the annotation $\texttt{check}_n^i \leftarrow \texttt{check}_{n-1}^i$ `or` $\texttt{reg}_x^i$`[15]` is required only once in a gate-level description even if it has to be checked in each cycle. Furthermore, the symbolic simulator can be extended to verify frequently checked properties without additional annotations. Extensions are facilitated by the fact that the information about each simulation step is available at the end of a path. Therefore, verification goals concerning intermediate *RegVals* or terms need *not* be checked *during* the path search since the information does not get lost, e.g., by rewriting terms.

The verification of reactive systems has to consider inputs of a circuit. The successive values of the inputs are *RegVals*, too. If the input pattern is known then the corresponding constants have to be assigned prior to each cycle to the *RegVal* of the input. Additional initial *RegVals* are introduced if the input value is unknown, i.e., a symbolic input value is used. A new initial *RegVal* is used for each input and each cycle. Note that those initial *RegVals* are identical in the specification and in the implementation. Intuitively, an input is modeled as a buffer which provides in each cycle the corresponding constant or a new symbolic value.

Example 2.8
Assume a gate-level description with an input `inport`*. The implementation at gate-level has to be simulated for three cycles to check equivalence to a specification (not shown in Fig. 2.7). The input is reset to "000" in the first cycle. The value of the input is arbitrary in the next two cycles. Fig. 2.7 shows the implementation to be simulated. Two initial RegVals* `in2` *and* `in3` *are assigned to* `inport` *before cycle two and three. The corresponding input values are used in the gate-level description since the occurrences of* `inport` *are accordingly indexed after each assignment.*

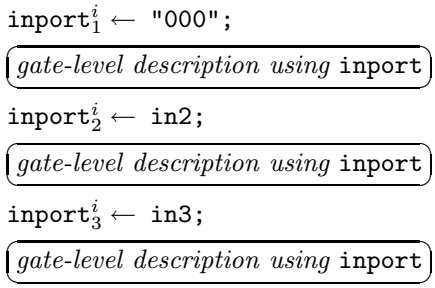


Fig. 2.7: Considering inputs during symbolic simulation

In the rest of this work, we assume, to facilitate the presentation, that the verification goal to be checked by the symbolic simulator is the computational equivalence of two descriptions.

2.8 Basic Algorithm of Symbolic Simulation

A brief overview of the basic simulation algorithm is given in the following. The implemented algorithm is presented more detailed in section 4.6.

The symbolic simulator is designed to compare two *acyclic* sequences. Frequently, the descriptions cannot be directly compared to demonstrate the verification goal as in the example of Fig. 2.2. Extracting the two sequences which demonstrate the verification goal is often simple. For example, two cycles have to be simulated symbolically to demonstrate the equivalence of the descriptions in the example of Fig. 2.3.

Algorithm 2.1 gives a simplified overview of the symbolic simulation algorithm which has been implemented iteratively for optimization, see section 4.6. The specification and the implementation are simulated in parallel. A case-split is performed when simulation reaches a condition C that cannot be decided in general but depends on the initial register values (lines 2 and 3). The information of the *EqvClasses* is used to decide conditions at branches consistently, i.e., to avoid unnecessary case-splits which lead to false paths. Note that *equivalence_check* is called recursively in line 3 with only those parts of *spec'* and *impl'* which are not simulated yet.

A complete path is found when the end of both descriptions is reached. The computational equivalence of the descriptions in this path is tested by checking whether the relevant final *RegVals* are in the same *EqvClass* (line 4). This test may fail since the equivalence detection during the path search is not complete to permit a fast symbolic simulation. Therefore, the *dd-checks* based on decision diagrams are used at the end of a path (line 5). They have to reveal whether

- computational equivalence is given in this path but was not detected (line 6, upper condition),
- a condition has been decided inconsistently due to the incomplete equiv-

alence detection on the fly (line 6, lower condition), i.e., a false path is detected, or

- a valid counterexample is found (line 7).

All relevant information of the path can be summarized in the last case to facilitate debugging. Our automatic verification process does not require insight of the designer into the verification process.

Algorithm 2.1 Simplified algorithm of the symbolic simulation

equivalence_check(spec,impl)

1. { Simulate *spec* and *impl* in parallel and perform intermediate *dd-checks* if necessary } **until**
   (a) a condition C is reached that cannot be decided in general but depends on the initial register and memory values, or
   (b) the end of both descriptions is reached.
2. **if** a condition C blocks **then**
3. `RETURN` $(equivalence\_check\,(spec', impl')\,|_{C=FALSE}) \wedge (equivalence\_check\,(spec', impl')\,|_{C=TRUE})$
4. **elsif** final values of registers are equivalent **then** `RETURN(TRUE)`
5. **else** perform *dd-checks*;
6. **if** (final values of registers are equivalent) $\vee$ (a condition has been decided inconsistently) **then** `RETURN(TRUE)`
7. **else** `RETURN(FALSE)`

Algorithm 2.1 is slightly modified if one of the descriptions is at gate-level rather than if both descriptions are at algorithmic-level or rt-level. Intermediate *dd-checks* are sometimes useful (line 1) in this case. Furthermore, the descriptions are not simulated in parallel. A complete path is searched instead in the specification before simulating the implementation at gate-level, see section 4.6.

Chapter 3

Related Work

Section 3.1 gives a brief review of symbolic simulation approaches referring also to the following sections. A recent symbolic simulation technique called symbolic trajectory evaluation (STE) is presented separately in section 3.2.

Sections 3.3 to 3.5 compare further formal techniques for sequential verification to our approach. Techniques based on validity checking, which are related to the early symbolic simulation approaches, are described in section 3.3. Section 3.4 discusses the use of theorem provers in our application area. Techniques relying on state space exploration are described in section 3.5.

A selection of semi-formal approaches, which use formal verification techniques, but do not focus on a complete verification, is presented in section 3.6. Consideration of memories in design verification is discussed separately in section 3.7 since it represents an important part of our symbolic simulator. Finally, section 3.8 summarizes the contributions of our work with respect to the approaches presented in the preceding sections. Techniques performing logic verification or combinational equivalence checking are not considered in the following since the purpose of our approach is sequential verification.

3.1 Review of Symbolic Simulation Approaches

Techniques using the principles of symbolic simulation have been used for many years. "Symbolic execution" as a technique for software verification was examined already in the 1970s [Kin75, Kin76, HK76, DK78]. Programs were executed using symbolic values for variables to demonstrate that they satisfy their specifications.

In the late 1970s ([Dar79] and [CJB79]), researchers at IBM applied the ideas of symbolic execution to hardware verification. [CJB79] introduced, according to [Bry90a], the term "symbolic simulation".[1] [CJB79] checked the equivalence of specifications and microcoded implementations, i.e., microprograms by executing both of them symbolically from corresponding states. Equivalence had to

[1] Darringer, working also at IBM, still used the term "symbolic execution" in [Dar79].

be demonstrated for all cases until the next defined point of correspondence was reached by using simplifiers or/and theorem provers.[2] Furthermore, [Dar79] describes an application to gate-level verification, e.g., comparing a two-bit counter to a corresponding gate-level description.

These techniques were continued by [Cor81][3] but they turned out to be not powerful enough at this time to reason about overall circuit behavior [Bry90a]. At each case-split, requiring a decision about a symbolic condition c, the *path conditions* of both branches were conjuncted with c and $\overline{c}$, respectively [Dar79]. The resulting expressions became too complex to be used efficiently [Bry90a] and the automatic symbolic manipulation techniques were not powerful enough [HS97]. Note that demonstrating equivalence of expressions had to be done using theorem proving techniques (requiring possibly user-interaction) if the previous simplifications were not sufficient.[4]

The following symbolic simulation approaches avoided building symbolic expressions, and used representations closely related to the underlying symbolic domain. For example, three possible values of a signal $\{0, 1, X\}$ can be encoded by two Boolean variables. The advantage of this representation compared to the previous approaches is that evaluation of functions, i.e., symbolic manipulation is better supported during simulation, especially by encoding and manipulating the symbolic signal values by *OBDDs* [Bry86].[5] These techniques were applied to switch-level verification [Bry85, BBB$^+$87, Bry90b, BF89, JG92].[6] *STE (Symbolic Trajectory Evaluation)* [SB95, BBS91] is an improved subsequent approach combining symbolic simulation with ternary modeling and using an *OBDD*-based encoding, too.

Symbolic evaluation was also used in theorem provers, e.g., it played a key role in the first version of the Boyer-Moore theorem prover [BM75], as recalled in [Moo98]. Section 3.4 compares "classical" theorem proving requiring mostly user interaction to our approach. Furthermore, a recent technique is discussed using a theorem prover as a tool to simulate symbolically an executable formal specification *without* requiring expert interaction.

The validity checking based approaches, described in section 3.3, are related to the early work of [Dar79, CJB79]. A formula is built implying the verification goal. Afterwards, this formula is demonstrated automatically by a validity

[2]Note that case-splitting is not automatic, since user interaction is possibly required to demonstrate equivalence for each case.

[3][Cor81] discusses how to simulate symbolically components written in the hardware description language ADLIB.

[4]The leaves of a "symbolic execution tree" [Kin75, Kin76, HK76, Dar79], produced by forking at each conditional statement, are closely related to our definition of a path. But decisions about conditions are considered in our approach by modifying *EqvClasses* instead of combining them by conjunction, see section 4.4.

[5]For example, two *OBDDs* are necessary for each signal to encode the three values $\{0, 1, X\}$.

[6]The earlier approaches used not yet *OBDDs* to encode the signal values. [JG92] examines particularly how to consider constraints, e.g., on the inputs during simulation.

checker. In contrast to the early approaches, the recent techniques cope with the complexity of the resulting expressions by using powerful validity checkers and/or restricting the application area, see section 3.3.

3.2 Symbolic Trajectory Evaluation

Symbolic Trajectory Evaluation (STE) [SB95, BBS91] is an efficient model checking approach which reasons about Trajectory Formulas, i.e., a restricted temporal logic which combines Boolean expressions and the "next-time" operator. STE verifies assertions $(A \Rightarrow C)$, i.e., properties. The system is simulated over the weakest trajectory for A which is a possible behavior of the model. Adherence of this trajectory to C is checked, which demonstrates that $A \Rightarrow C$ holds. STE operates on *symbolic* values, parameterized in terms of a set of Boolean variables which encode a symbolic value for different operating conditions. For example, the behavior of an inverter can be specified by `[in is` a $\Rightarrow$ `N(out is` $\neg a$`)]`. STE uses a lattice representation for the circuit states. For example, for switch level verification (from where STE grew out) the values representing the lattice $\{X, 0, 1, \top\}$ are used.[7] Usually two *OBDDs* are used to represent each symbolic node value. [KG99] provides a good introduction to STE. A historical survey is given in [HS97].

An advantage of STE compared to other model checking techniques is that it is sensitive to the *property* to be verified rather than to the state space. It has been successfully applied to the verification of large memory arrays (e.g., [PB99, WAK98, HS97, PRBA97, PRBB96]) at transistor-level. Symmetries of data and structure are used during verification. Properties of datapath components like multipliers or systolic arrays [HS97] and of the Intel™ instruction marker [AJS98] have been verified *with user interaction* using Voss[8] which combines STE and theorem proving. The verification of complex industrial floating-point designs was done with Forte, an evolution of Voss, but required significant human effort [OZGS99, AJK+00]. A decomposition of the verification task into smaller parts by data space partitioning is used in [AJS99] to allow an automatic verification of floating-point units and of an Intel™ instruction marker using Voss. A parametric representation is used to encode the data space constraints in the different case splits provided by the user. The approach makes use of the fact that the symbolic simulation technique applied is faster on a constrained data space. A methodology for hardware verification using Forte (including STE) is surveyed in [AJM+00].

Although well suited to verify functional properties of *data intensive* parts or components, an application of STE to the verification of complex *control* systems with data operations against a specification at higher level is not clear due to

[7] X represents the unknown and $\top$ the "overconstraint" value.

[8] Also denoted as VossProver.

the representation of symbolic values by decision diagrams. Furthermore, the restricted logic constrains the applicability.[9]

3.3 Validity Checking Based Techniques

Techniques based on automatic validity checking have been successfully applied to equivalence checking of descriptions at behavioral rt-level and structural rt-level. They divide the verification problem into two steps:

- a formula F is built which implies that the verification goal is satisfied, i.e., $F \Rightarrow verification\ goal$, and then
- a validity checker demonstrates automatically that $F \equiv true$.

Some verification problems can be reduced to a formula in which all functions except equivalence and the Boolean operators are considered as uninterpreted functions. Ackermann [Ack54] demonstrated such a reduction to formulas of the theory of equality without interpreted functions while preserving validity.[10]

For many verification problems, it is not sufficient to have only a decision procedure for uninterpreted equality, e.g., because bit-vector arithmetic is required to demonstrate the verification goal. The problem is to consider different decision procedures of the component theories like arithmetic, arrays etc. Two approaches of decision procedures for *combinations* of theories have been pioneered in the seventies [CLS96]. Nelson and Oppen [NO79, NO80] combine theories by iteratively propagating equalities between different decision procedures. A practically more efficient procedure developed by Shostak [Sho84, Sho79] combines the simplifiers of different theories into a single decision procedure. A good description of Shostak's algorithm is given in [CLS96]. Note that decision procedures are also used in theorem provers (see section 3.4), e.g., PVS uses Shostak's algorithm [ORSvH95].

A prominent example for applying automatic validity checking to hardware verification was presented by [BD94]. They were first to propose a technique to generate a logic formula that is sufficient to verify a pipelined system against its sequential specification. This approach has also been extended to dual-issue processors [JDB95], super-scalar architectures [Bur96, WB96][11], and with some limitations to out-of-order execution by using incremental flushing [SJD98, JSD98].

[9] Assertions about the correct effects of single instructions of a small 16Bit-CISC-processor have been manually derived and verified in [BB94] using STE (although the term STE is not used in [BB94], see [SB95]).

[10] Ackermann's formulas include also existential and universal quantifiers, which are not considered in the following.

[11] [WB96] provides a formal verification (using HOL) of the decomposition theory given in [Bur96] for superscalar architectures.

SVC (the *Stanford Validity Checker*) [BDL96, BDL98, JDB95] was used to automatically verify the formulas. SVC is a proof tool using an algorithm similar to Shostak's decision procedure. SVC requires also for each theory to add that functions are canonizable and algebraically solvable, because every expression must have a unique representation. The tool can fail to prove equivalence if a design is transformed by using theories, that are not fast to canonize/solve or that are not supported.

[BDL98] describes the extension of SVC with bit-vector arithmetic (inspired by the work in [CMR97][12]). Verification of bit-vector arithmetic is often required to prove equivalence in control logic design and is fast using SVC if expressions can be canonized without slicing them into single bits. Otherwise computation time can increase exponentially. Our approach does not generally canonize expressions. Only if corner-cases of equivalence have to be detected to demonstrate the verification goal, then formulas are constructed *using previously collected information* and are checked using *vectors* of *OBDDs*. The efficiency of vectors of *OBDDs* in our application area is compared with SVC and **BMDs* in section 6.3. Verification of memories using SVC is discussed in section 3.7.

SVC is not an uninterpreted approach since a selection of functions is interpreted by SVC. Only *uninterpreted* functions with the exception of memory-operations[13] are used by [VB00, BGV99, VB99a, VB99b] for equivalence checking of *high-level descriptions* of processors against instruction set specifications. Two abstract formulas are built, similar to the approach of [BD94, Bur96], and compared using *OBDDs*. An extension which exploits positive equality makes verification of pipelined [BGV99, VB99a] and superscalar [VB00, VB99b] processors feasible in seconds, a significantly inferior verification time compared to [BD94, Bur96]. This extension considers that some comparisons only occur in monotonically positive formulas, i.e., they do not appear in the scope of a logical negation. The approach is well suited for the given verification examples. The pipelined or superscalar architectures could be derived from the sequential specifications mostly by scheduling and without considering bit-vector arithmetic operations, see also section 7.1. The approach is limited to such verification examples which do not require an interpretation of functions.

[LO97, LO96] propose an approach for pipeline verification different to the technique of [BD94]. The pipeline verification problem is decomposed in smaller, simpler steps by "unpipelining" successively the implementation. The result is a sequential description. The formulas implying correctness of the different steps were checked using SVC. Their specialized approach relies on a standard design style and requires that different parts of the pipeline stages can be extracted.

[12] [CMR97] developed a decision procedure for fixed-size bit-vectors. The main difference in [BDL98] is that "bitplus"-expressions, i.e., addition of bit-vector variables modulo the bit-width, are used as internal representation in SVC to increase the range of examples which can be verified automatically.

[13] `read`- and `write`-operations are interpreted as described in section 3.7.

Techniques generating a single formula for the verification problem, which is verified afterwards with a validity checker like SVC, do not distinguish explicitly the different intermediate symbolic values of the registers: an assignment is considered by using the symbolic term assigned whenever the register is used afterwards. This can lead to term-size explosion and/or case-explosion for sequential verification, especially at structural level. For example, a big ROM or the implementation of the control part by multiplexers has to be considered as argument after each sequential step and the corresponding expression may not be simplified on the fly. In general, an application to *gate-level* descriptions is not possible since in each step the whole gate-level expression has to be substituted and the resulting formula cannot be checked even with support by bit-vector arithmetic decision procedures. Furthermore, the information about the sequential behavior gets lost and the debugging information of the counterexample is restricted to an expression in the initial register values. Therefore, we do not replace in our approach the intermediate register values but distinguish them only by indices, see section 2.4.

3.4 Theorem Proving Techniques

Theorem proving techniques rely on expressing the system and the desired behavior in the formal language of the theorem prover based on some mathematical logic. The process of finding a proof of a property from the axioms of the system is called theorem proving [CW96]. Numerous theorem provers exist, demonstrating the interest in these techniques.[14] Some well-known theorem provers are ACL2 [KM97, BKM96] and its predecessor Nqthm [BM97, BM79], PVS [ORSvH95, ORS92], or HOL [GM93].

Theorem proving techniques have been successfully applied to complex hardware verification problems. Prominent examples are the verification of the FM9001 microprocessor [BHK94] using Nqthm, of the Motorola CAP processor [BKM96] using ACL2, and the verification of the AAMP5 processor [SM95b, SM95a] using PVS. As well as for those examples, theorem proving techniques often require extensive user guidance from experts to find the proof. For some verification problems, the need for user-interaction can be limited by using application specific strategies. For example, [HSG98, HSG99] proposed an interesting technique to decompose the verification of processors with pipelining [HSG98] and out-of-order execution [HSG99] against sequential specifications in sub-proofs.[15] The approach uses for each unfinished instruction a completion function describing the *effect* of completing the instruction. Note that the need for user guidance remains especially for less regular designs.

In summary, theorem proving techniques using general algorithms have a larger

[14] [Bow00] provides a good list of links to theorem proving tools.

[15] PVS is used to carry out the proofs.

application area than our symbolic simulation approach, but they require significant user interaction for our verification problems. Our method is automatic.

An approach to use a theorem prover to simulate symbolically an executable formal specification *without* requiring expert interaction is described by [Moo98] using ACL2. Related is the work in [Gre98], where pre-specified microcode sequences of the JEM1 microprocessor are simulated symbolically using PVS. Expressions generated during simulation are simplified on the fly. Multiple numerical simulation runs are also collapsed, but the intention of [Moo98] is completely different since concrete instruction sequences at the machine instruction level are simulated symbolically. Only a *fast* simulation on *some* indeterminate data is possible. Our approach checks equivalence for *every* possible execution, e.g., not only some data is indeterminate but also the entire control flow. Indeterminate branches would lead in [Moo98] to an exponential growth of the output to the user. Furthermore, insufficient simplifications on the fly can result in unnecessary case splits or/and term-size explosion. The approach of [Moo98] provides a *fast* simulation on some indeterminate data, e.g., for debugging a specification. If simulation can run automatically (i.e., without additional information provided by the user) then simulation speed is significantly higher than in our approach.

3.5 Techniques Relying on State Space Exploration

All techniques which depend on state space exploration face the problem that the number of states grows generally exponentially with the number of storage elements, which is known as the *state explosion problem*. This remains an important limitation even if states and transition relation are represented symbolically by decision diagrams. The idea of symbolic state space representation has already been applied by [CBM89b, BB94] for equivalence checking or by [BCM+92, BCMD90, BCL+94] for traversing automata for symbolic model checking[16]. State explosion occurs particularly if the system being verified has different components that can make transitions in parallel [CGP99]. The number of global states may grow exponentially in this case with the number of processes. Another reason for large state spaces are data structures with many different values, e.g., the data path of a processor [CGP99].

The equivalence of two deterministic finite state machines (*FSM*) can be demonstrated by building the *product machine*. The inputs of the machines are connected. The output of the product machine indicates pairwise equivalence of all the outputs of the two machines. The two *FSMs* are equivalent iff for any transition reachable from the initial states the product machine produces the output *true*, i.e., the outputs of the two machines are identical. The verification faces the state explosion problem since the transitions from all reachable states

[16] See [CGP99] for an overview.

have to be considered. Note that in the case of incomplete specified systems, state traversal is not applicable.

Generally, equivalence checking techniques that verify the product machine avoid an explicit enumeration of states, just like symbolic model checking methods.[17] State space and transition relation are represented symbolically by decision diagrams, usually *OBDDs*. State traversal for equivalence checking using such a symbolic representation has been described first in [CBM89b, CBM89a, CBM90]. Symbolic model checking also depends on the complexity of the state space, since the verification is done by iteratively traversing at least parts of the state space.

Several techniques to tackle the well-known state explosion problem have been proposed. Three examples are given in the following. A survey of other approaches to the state explosion problem is given in [CGP99].

An abstraction method which converts the state space to a reduced state space is described in [ID96]. Reversible state generation rules are identified to collapse multiple states into one abstract state. The disadvantage of this technique is that the rules for protocol verification reported in [ID96] are derived *manually* and identification of such rules may be difficult for other designs.

[AGM96] describe an alternative to state traversal for equivalence checking if the specification *FSM* has the *Complete-1-Distinguishability* property, i.e., each state can be distinguished from all others by an input sequence of length 1. For example, a Moore machine has this property, if the outputs of all pairs of distinct states are different. In this case, only 1-equivalence (i.e., a single step) has to be verified. The approach is restricted to circuits for which the property above holds. Otherwise, internal latches have to be denoted as "primary-pseudo-outputs"[18] which restricts synthesis significantly.

[CCPQ99] address the problem of silent paths. No activity under constant inputs can be observed on those paths, e.g., a counter is started and a single output indicates overflow after n-steps. The idea is to "jump" over the states with identical output behavior, i.e., overflow is reached within one step. Their method relies on *OBDD*-based *FSM*-representations of the circuits, too, and has the same limitations concerning the state space representation as described above. The technique has been basically applied in [CCPQ99] to *speed up* symbolic simulation of counters.

In summary, various techniques exist to tackle the state explosion problem which allow pushing the limit further but either do not provide a general solution for fast automatic traversal of large circuits or their area of application is restricted.

[17] [SD98] found that for some examples an explicit enumeration of the states can save up to a factor of 50 or more memory space if the BDD is close to worst-case behavior as for directory-based cache coherence protocols.

[18] They allow to distinguish states that have the same values on the "real" outputs.

3.6 Semi-Formal Approaches for Fast Falsification

Numerical simulation with test-vectors is incomplete since only a non-exhaustive set of cases can be tested. Several promising approaches exist to speed up numerical simulation which permit a faster and more efficient debugging but do not overcome the case explosion problem.

The techniques discussed as examples in the following are denoted *semi-formal* approaches since they use formal verification techniques, but do not focus on a *complete* verification. Other techniques like numerical simulation are combined with formal methods to speed up verification, e.g., by aggregating different cases or by applying various techniques in a heuristic manner. Verification (or validation) remains incomplete although more cases are considered than without formal methods. Completeness is sacrificed either to allow a faster falsification, e.g., by aggregating simulation runs or to permit validation of larger circuits.

Three related heuristics for verification are proposed by [WDB00, BDQ99, GAK99]. Numerical and symbolic simulation are combined in [BDQ99]. In each clock cycle, parts of the inputs are tied automatically to constants (as in numerical simulation) while others get symbolic values. Graph-explosion of the *OBDDs* is avoided because of the constant inputs while the number of test vectors simulated in one time unit is significantly increased compared to numerical simulation.

[WDB00] focus on system-level design integrating several components. Formal verification often fails at this level due to the size of the design. An *automatic* case-splitting algorithm also ties symbolic variables to constants to control graph size at the expense of increased simulation time. Furthermore, *approximate* values are used on internal nodes, i.e., the function representing a node value can result not only `0` or `1`, but also `X`. Nodes not affecting the functionality in the current case according to a given test are set to `X` to minimize the decision diagram representation. A heuristic is used to identify variables for case-splits and to guide approximation.

The objective of [GAK99] is to find efficiently counterexamples to safety properties by using iteratively numerical simulation, *OBDDs*, and *ATPG*. The circuit is simulated and nodes which remain unchanged are remarked. A heuristic "solver" uses *OBDDs* and *ATPG* techniques with a defined computation limit to generate inputs enabling transitions which have not been taken yet. These results are used to guide numerical simulation in the next step. Especially the third approach is related to our symbolic simulation by applying alternating different techniques. However, the intention of the three approaches is different since they sacrifice in their heuristics completeness of the verification process in order to allow a fast "falsification" without guaranteeing that corner cases are considered. Note that the approach of [Moo98] described in section 3.4 also provides a *fast*

simulation on *some* indeterminate data using ACL2. This can be useful, e.g., for debugging a specification.

Another hybrid approach mixing numerical simulation and formal methods is proposed by [GMA97] to overcome the state explosion problem. A smaller test model is derived from the design which can be handled by a formal verification technique. This technique generates test-vectors for numerical simulation of the real circuit which should maximize coverage of design errors. Deriving the test model is non-trivial and complete coverage of the generated test-vector set is only given on some assumptions.

[CRS98, CRS99] propose a technique for fast error detection on large designs. A genetic algorithm is used to provide only as soon as possible a counterexample to sequential equivalence if one exists. The user has to pre-define *checkpoints* which are assumed to be coupled. The population is represented by different input sequences. The fitness of each sequence depends on differences at the checkpoints and their propagation in the two circuits since the objective is to find a sequence propagating a difference to the outputs. The heuristic does not guarantee to find an existing counterexample and a positive confirmation of equivalence is not possible.

Although the approaches described above do not provide a complete verification, they can be helpful for fast "falsification" of a design, i.e., to find quickly "bugs" or to improve the design. Furthermore, if formal verification approaches fail to demonstrate the verification goal, e.g., because the circuit is too large, then these techniques can increase protection against implementation errors.

Note that the paths to be simulated symbolically can be restricted in our approach, for example, by annotating the initial description. This allows a selective faster verification of only the cases considered by these paths.

3.7 Verification of Memories

Verification tools must often cope with large memory sizes and symbolic addressing. The verification problem can be divided into two parts if memories are described as separate blocks or units:

- verification of the memory block itself, i.e., whether the structural implementation of the block meets the requirements. For example, STE has been successfully applied to the verification of large memory arrays (e.g., [PB99, WAK98, HS97, PRBA97, PRBB96]), see section 3.2;

- interaction of the memory with the rest of the system; abstraction of the implementation details of the memory block facilitates the verification of the entire circuit; however, the abstract model has to capture the functionality of the memory, e.g., two `read`-operations with the same address result in the same value if there is no intervening `store`-operation. Oth-

erwise verification of the entire circuit can produce false negatives or false positives.

Various representations of memory operations have been proposed for formal verification of digital circuits. States are often represented by decision diagrams by techniques relying on state space exploration, e.g., [BCMD90, BB94]. This permits the representation of a register file but not of a large data memory due to the sensitivity to graph explosion, see section 3.5.

SVC (see section 3.3) verifies automatically formulas which can contain the two array operations `read` and `write` to model memory operations.[19] Verification of control logic is possible using SVC if the verification task can be reduced to a formula which is sufficient to demonstrate the verification goal. Relationships of *memory operations* are revealed by SVC basically by case analysis. A `read`-operation $\texttt{read(write}(s, aW, v), a1\texttt{)}$ after a `write`-operation is rewritten to $\texttt{ite}(a1 = aW, v, \texttt{read}(s, a1))$. A case analysis is required to prove that $\texttt{read(write}(s, aW, v), a1\texttt{)} = \texttt{read(write}(s, aW, v), a2\texttt{)}$ follows from $a1 = a2$. The case analysis guarantees the functional consistency of the abstract memory model. A similar way of abstraction and reasoning is used by [VB00, VB99b] et al,[20] see also section 3.3.

Our approach avoids case analysis *on memory operations*. Equivalences of memory operations as in the example above are detected in a different manner during simulation. Rewriting and case analysis can become also not practicable in a validity checker if memory operations cause too many case splits. This can be the case, for example, if operands are read repeatedly from a memory and the result is written back. Consider a simple architecture, where an instruction with two source- and one destination-address is read from an instruction memory. The source values are read from data memory, they are added, and the result is written back. Finally, the program counter is incremented and the next instruction is fetched. Equivalence checking of the data memory after, e.g., six instructions requires already 11,868,920 case splits using SVC (4396s on a 300 MHz Sun Ultra II), if we reverse the order of the first two instructions addressing distinct places in the data memory. Our approach avoids these case splits.

[Moo98] uses ACL2 to simulate symbolically executable formal specifications, see section 3.4. Memories are modeled as lists of symbolic values which represent the memory contents, i.e., the length of the lists grows with the memory size. This explicit modeling allows no efficient automatic reasoning about symbolic values of address registers, since, e.g., a store-operation with a symbolic address can change *any* memory place. As discussed in section 3.4, the intention of [Moo98] is completely different since a *fast* simulation on *some* indeterminate data is provided.

[19] Our model of memories is similar, see section 4.1.5.

[20] Although the approach of [VB00, VB99b] is slightly different with respect to the replacement of uninterpreted functions by domain variables.

3.8 Contribution of this Work

Table 3.1 summarizes the main advantages and limitations/inconveniences of the techniques discussed in the preceding sections compared to our approach. The most common distinguishing feature is the application area of our symbolic simulator described in section 2.2. The main contributions of our approach are:

- interpreted sequential verification at different levels of abstraction as demonstrated by experimental results:
  - automatic *sequential* verification of gate-level results of a commercial synthesis tool against a behavioral or structural specification at rt-level, see [Rit00] and section 7.3;
  - automatic *sequential* equivalence checking of two descriptions at rt-level at different levels of abstraction, i.e., structural descriptions with implementation details can be compared with their behavioral specifications, see [REH99] and section 7.2;
- the flexible use of an open library of different equivalence detection techniques in order to find a good compromise between accuracy and speed. Additional equivalence detection algorithms can be integrated without requirements like canonizability of functions;
- an effective combination of symbolic simulation and decision diagrams to detect corner-cases of equivalence;
- equivalence checking of descriptions with complex reorderings of memory operations, see [RHE99] as well as section 5.9 and 7.1;
- a verification which is independent of the specific synthesis tool and copes also with manual modifications of the designer;
- a good debugging support.

These results are made possible by the essentials described in section 2.1 which distinguish our symbolic simulation approach.

The objective to use our symbolic simulator for property verification as described in section 2.7 is not considered in Table 3.1 and above, since no experimental evidence exists with the exception of first results concerning register binding verification. The same holds for verification at algorithmic level.

	Advantages compared to our approach	**Limitations/Inconveniences compared to our approach**
STE and previous approaches	• property verification possible (model checking) • verification of large/complex memories and data components • combination with theorem proving techniques possible	• application to complex *control systems* (without user-interaction) ? • application at higher level of abstraction ?
Validity checking based techniques	*SVC based techniques* • faster if interpretation is sufficient • verification of complex processor examples (number of paths to verify) at rt-level possible *Uninterpreted approaches* • very fast on problems requiring no interpretation of functions • significantly faster even than SVC on those examples	• interpretation has to be sufficient ⇒ requirements on new theories • possible term-size/case-explosion • limitations of bit-vector arithmetic • application at gate-level ? • consideration of memory operations • information of counterexample • uninterpreted approaches: limited to problems requiring no interpretation of functions
Theorem proving techniques	*General* • larger application area • cope with very large and complex designs *Used as symbolic simulation tool* • fast symbolic simulation for debugging	*General* • not automatic ⇒ require often extensive user guidance from experts *Used as symbolic simulation tool* • control flow not indeterminate in order to do without user guidance • symbolic addressing of memories ?
Techniques relying on state space exploration	• property verification (model checking) • reason about infinite sequences • state traversal can be faster than symbolic simulation • interpretation of functions is irrelevant once the transition relation is extracted	• state explosion problem • consideration of memories • incomplete specified systems
Semi-formal approaches	• fast "falsification" or debugging • application to large designs ⇒ increase protection against implementation errors	• incomplete verification ⇒ consideration of corner cases not guaranteed • heuristic approaches ⇒ coverage ?

Tab. 3.1: Comparison of the symbolic simulation approach to other techniques

Chapter 4

Symbolic Simulation Procedure

Modifications of the data structure before symbolic simulation are described in section 4.1. Section 4.2 discusses the strategy for invoking the equivalence detection. Section 4.3 describes how the results of the equivalence detection are notified using *EqvClasses*.

The evaluation of conditions during symbolic simulation is presented in section 4.4. Section 4.5 gives two examples for symbolic simulation runs to illustrate the approach. Finally, section 4.6 presents the actual implementation of the symbolic simulation algorithm introduced in section 2.8.

4.1 Preparing the Data Structure for Symbolic Simulation

The symbolic simulator requires some substantial modifications of the initial data structure which are performed in a pre-processing step. Finite sequences have to be generated from the descriptions to be verified since the number of *simulation* steps must be finite. Section 4.1.3 demonstrates that the verification problem can be reduced for many cyclic designs, e.g., pipelined machines to the equivalence check of acyclic sequences.

The input language is briefly described in section 4.1.1. Section 4.1.2 gives an overview of the compilation tools used. The main transformations are presented in section 4.1.3 to 4.1.5. Additional transformations are reported in appendix 9.1 to 9.3.

4.1.1 Input Language

The experimental hardware description language *LLS* (*Language of Labelled Segments*) is used as input language for our symbolic simulator. A detailed description of *LLS* is given in [Hin98b], see also [EHR98, Hin00].

A frequently used universal language as VHDL, which was mainly developed for simulation purposes, has the disadvantage that it lacks standardized formal semantics. Therefore, its applicability to formal synthesis and verification is limited. Synthesis tools support only subsets of VHDL.

LLS, a further development of *SMAX* (*SMall and AXiomized*) [Eve91, ES92], possesses a formal semantic which allows to support formal synthesis and verification. It is an experimental, axiomatized hardware description language which permits to describe a closed, deterministic, synchronously parallel transition system. *LLS* is mainly intended to represent systems at rt-level or algorithmic-level, but allows also a description at gate-level. Extended FSMs (EFSMs), which are a common concept in many approaches, can easily be represented in *LLS*. Figure 4.1 gives an example adapted from [RJ95] (the description calculates $a \cdot b \ mod \ n$) in extended FSM notation and the corresponding textual *LLS* representation. The symbol "-" in a condition denotes that the transition is taken in any case. The same symbol as action represents a `STALL`, i.e., the register values remain unchanged. Labels like `L0` correspond to control states, and are used to guide

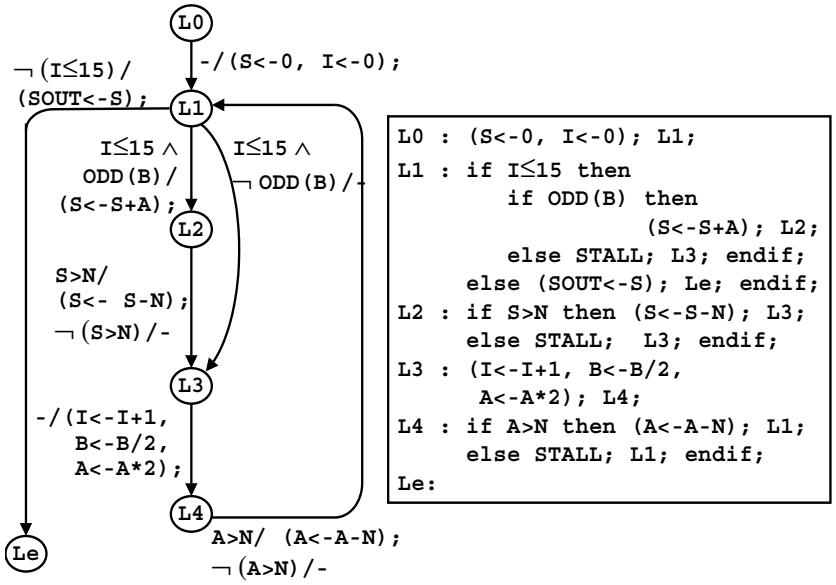


Fig. 4.1: Extended FSM and corresponding *LLS* description. Taken from [EHR99]

the flow of control. An initial label (`L0` in Fig. 4.1) has to be identified for each description. A *LLS* description consists of a number of segments of the form `L: B` where `B` is called the segment-body associated with label `L`. The labels occurring in the segment-body are called exit labels, and are used to specify the flow

of control; e.g., L2, L3 and Le are the exit labels of segment L1 in Fig. 4.1. The data operations are specified in the segment body B. Assignments to a variable like x←y are called transfers. Parentheses enclose synchronous parallel transfers, e.g., (x←y, y←x) exchanges the contents of x and y in a single step. The sequential composition operator ";" separates consecutive transfers, see for example Fig. 4.2. The content of the variable y after the execution of the segment

```
M0:   (x←a+b);
      (y←x-1); M1;
```

Fig. 4.2: Example of sequential transfers in *LLS*

body of M0 is a+b-1 and control is transferred to M1.

Branches are realized by *if-then-else*-clauses. Cyclic behavior has to be modeled by branches and exit labels since no explicit loop-construct is provided.

Compilers from a subset of *LLS* to VHDL, from a subset of VHDL to *LLS*, and from a subset of C to *LLS* exist and are presented in the following section.

4.1.2 Overview of Compilation Tools

Two sets of compilers are used for pre-processing, Fig. 4.3 gives an overview. The first set is not specific to the symbolic simulator, i.e., those compilers are shared with other tools or applications. They translate descriptions between the intermediate data format *IDS* (Internal Data Structure using GNU Common Lisp commands) and other representations:

- the *LLS* compiler [EHR98, Hin98b, Hin00] translates between the textual representation *LLS* and *IDS*;
- the *C2LLS* compiler [Lev00] supports a subset of ANSI C; it generates first a C description similar to the *LLS* format which is used to derive a *LLS* description;
- the *SYN2IDS* translator[1] transforms synthesis results of the Synopsys® Design Compiler™ using the Alcatel™ MTC45000-library in VHDL to *IDS* format; it has been implemented to allow a *sequential* verification of the synthesis results. The compiler is described in appendix 9.4;
- the *IDS2VHDL* translator [Hin00] transforms an *IDS* description into a VHDL design. Since memories are modeled as arrays in *LLS*/*IDS*, which are not suitable for synthesis, they are described structurally in VHDL by generating the corresponding address-, data-, and control-signals to a standard memory block, see appendix 9.4.

[1]The term "translator" is used instead of "compiler" since the tool only transforms the data format. For example, a syntax check (like in the *LLS* compiler) is not provided. The same holds for the *IDS2VHDL* translator, see below.

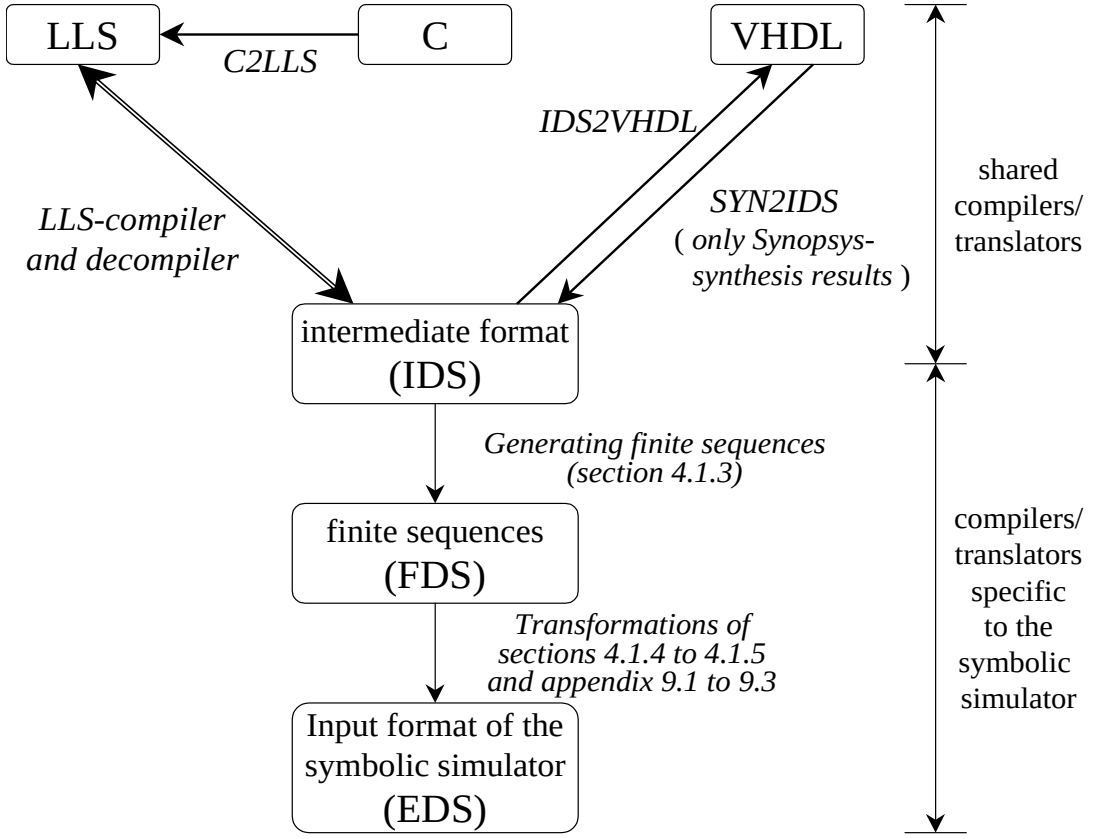


Fig. 4.3: Overview of compilation tools

The *IDS* data structure is also used for other tools, e.g., automatic pipeline construction [HER99, Hin00] or verification of register-binding [BRHE00] and is not adapted to symbolic simulation. Therefore, two compilers specific to the symbolic simulator have been developed. The first one generates finite sequences as described in section 4.1.3. The second one performs all other transformations necessary for symbolic simulation which are presented in section 4.1.4 to 4.1.5, and appendix 9.1 to 9.3.[2]

Note that all transformations and modifications are achieved automatically. Only the generation of the finite sequences requires in some cases an annotation in the initial description which is discussed in section 4.1.3.

4.1.3 Generating Acyclic Sequences

Symbolic simulation is able to compare only terminating descriptions, i.e., descriptions which consume only a finite number of computation steps and which have to consist, therefore, of an acyclic sequence of statements. However, for

[2]The designations *FDS* format (*Flushed Data Structure*) for the format after the first compiler and *EDS* (*Equivalence-checker Data Structure*) for the input format of the symbolic simulator are used for historical reasons; the symbolic simulator was first applied to equivalence checking of systems with pipelining.

many cyclic designs the verification problem can be reduced to the equivalence check of acyclic sequences. Determining those sequences requires only an insight of the user in his own design but not in the automatic verification process. Generating acyclic sequences consists in

- unrolling finite loops, and
- breaking infinite loops, which are described either explicitly (e.g., in an algorithmic description) or implicitly (e.g., description of a processor on which a program with an arbitrary number of instructions can be executed).[3]

Finite Loops

Loops with a *limited* number of iterations can be unrolled if the *upper limit* of iterations is known: an *if-then-else*-clause with the loop body in the **then**-branch is replicated according to the upper limit. The *if-then-else*-clause tests the loop condition, i.e., the corresponding loop body is only simulated if the condition is true; otherwise symbolic simulation reaches the "empty" **else**-branches, i.e., the additional cycles are ignored (`STALL` signifies that the register values remain unchanged). Note that only the upper limit of iterations has to be known. The number of iterations may vary depending on the path, see Example 4.1.

Example 4.1
Fig. 4.4 (a) shows a loop in pseudo-code, which would be implemented in LLS using branches and exit labels. The description to simulate symbolically is given

(a)
```
if a=b then i←0;
       else i←3;
while i<5 do
       res←y+res;
       i←i+1;
od
```

(b)
```
if a=b then i←0;
       else i←3;
if i < 5 then res←y+res; i←i+1;
         else STALL ;;change nothing
if i < 5 then res←y+res; i←i+1;
         else STALL ;;change nothing
if i < 5 then res←y+res; i←i+1;
         else STALL ;;change nothing
if i < 5 then res←y+res; i←i+1;
         else STALL ;;change nothing
if i < 5 then res←y+res; i←i+1;
         else STALL ;;change nothing
```

Fig. 4.4: Unrolling of loops with upper limit

in Fig. 4.4 (b). The upper limit of iterations is 5, but the loop may terminate after 2 iterations. Three "empty" **else***-branches* (`STALL`) *are simulated in this case. Note that loop termination is determined in both cases automatically by detecting equivalence of* `i<5` *and* `0` *(false).*

[3]Note that explicit loops are also modeled by branches and exit labels in *LLS* since no explicit loop-construct is provided.

Infinite Loops

Many cyclic designs contain an infinite loop, e.g., fetching and executing repeatedly an instruction on a processor. Those infinite loops have to be "broken" since otherwise simulation does not terminate on all possible paths. Reducing the verification problem for those designs to a comparison of two finite sequences is often possible by simply comparing a finite number of executions of the loop bodies in the specification and in the implementation:

Example 4.2
A behavioral specification is given, where the execution of one instruction takes only two cycles. The implementation is a microprogram-architecture which executes an instruction in 8 to 10 cycles depending on the instruction. The execution of instructions is not overlapped.

The acyclic sequences to be compared in this example are the execution of one instruction *in the specification and in the implementation. If the final values of the registers are the same for all acceptable initializations then an arbitrary sequence of instructions produces the same results as well, i.e., the descriptions are computationally equivalent. Note that arbitrary values have to be assumed for additional registers in the implementation.*

The finite sequence, which describes the execution of one instruction in the behavioral specification can be often detected automatically: all exit labels (see section 4.1.1) which have not occurred along the path of execution, are replaced iteratively by the corresponding segment body. The instruction is completed if a label is reached which has been already used. Alternatively, the user lists explicitly the sequence of labels which represent the execution of an instruction.

The description of the structural implementation represents only one cycle. This description has to be replicated 10 times in order to consider the maximum number of cycles to be simulated symbolically. An additional comment of the designer has to prevent the simulation of redundant cycles for shorter instructions with only 8 or 9 cycles. This is done by simply introducing a flag, which signals whether an instruction has already been started and which is evaluated before a new instruction is started.[4] *The realization of this short comment in the LLS language is given in appendix 9.5.*

Note that the information provided by the user concerns only the functionality of the design and can be provided without knowledge about the verification process.

The execution of instructions in Example 4.2 is not overlapped. Therefore, equivalent states have to be reached in both descriptions after each instruction.

Symbolic simulation copes also with overlapped execution to demonstrate computational equivalence. The finite sequences are straightforward to construct if the loop bodies of the specification and of the implementation are identical, or

[4] This test is similar to the unrolling of the finite loop, see Example 4.1. The flag corresponds to the loop condition.

if n iterations of the specification loop should produce the same results as m iterations of the implementation loop.

Example 4.3
Two structural descriptions of a microprocessor are compared. The execution of an instruction takes 3 cycles in the sequential specification. The implementation fetches and executes two *instructions in 3 cycles without data or control hazards. The loop is infinite, i.e., computational equivalence has to be demonstrated for arbitrary instruction sequences. But it is sufficient to compare 6 cycles of the specification to 3 cycles of the implementation.*

Comparing a distinct number of executions of the loop bodies as in Example 4.3 is not sufficient if the loop bodies overlap differently in the specification and in the implementation. An important class of verification examples where such an overlapping has to be considered is the equivalence check of a pipelined processor and the corresponding sequential specification.

Example: Verification of Systems with Pipelining

Pipeline verification is used in the following as an example to demonstrate how the verification problem can be reduced to a comparison of two finite sequences even if loop-unrolling or matching only parts of the infinite loops in the specification and in the implementation cannot be applied in a straightforward way.

Example 4.4
An implementation of the DLX-architecture [HP96] with a five stage pipeline is compared to the instruction set architecture (ISA) of the DLX, which is modeled by a sequential description.

The execution of instructions are overlapped in architectures with pipelining to optimize the throughput. Therefore, the equivalence of a system with pipelining and of a sequential specification cannot be demonstrated by comparing the execution of a single instruction, since the overlapped preceding or succeeding instructions modify the state of the processor. Burch and Dill [BD94] proposed an approach which allows to verify a pipelined system against its sequential specification by using the flushing property of the pipelined design (see below). This approach has also been extended to the verification of dual-issue and (with limitations) super-scalar architectures [JDB95, Bur96, WB96].

Pipelined processors typically have an external input which forces the processor to continue the execution of instructions already in the pipeline while not fetching new instructions which is called *stalling* the processor. After having stalled a processor for a finite number of cycles, all remaining instructions are completed and the pipeline is empty which is referred to as flushing the processor.

The equivalence check can be reduced to a comparison of two sequences:

- starting one instruction in the pipeline and flushing afterwards;

- flushing the processor and executing the last instruction on the sequential processor.

In the first case the last instruction is executed on the pipelined system while in the second case it is executed on the sequential processor of the specification.

Example 4.5
Fig. 4.5 shows the principle for a 5-stage DLX-Pipeline. Hazards are neglected for simplicity. Each instruction consists of five stages `IF` *to* `WB`*. Fig. 4.5 (a) and Fig. 4.5 (b) both describe the end of the execution of an arbitrary program. The last instruction is also started on the system with pipelining in Fig. 4.5 (a) while it is executed on the sequential processor in Fig. 4.5 (b). Because the dotted areas on the left side are identical, it is sufficient to compare the sequences on the right side:*

- *the last instruction is started in the pipeline and then the flushing takes four cycles;*
- *the immediate flushing of the pipeline takes four cycles; the last instruction is executed on the sequential processor.*

The processor is in this example *in the full pipeline state at the beginning of both sequences. Note that other states, e.g., due to previous hazards have to be considered, too.*

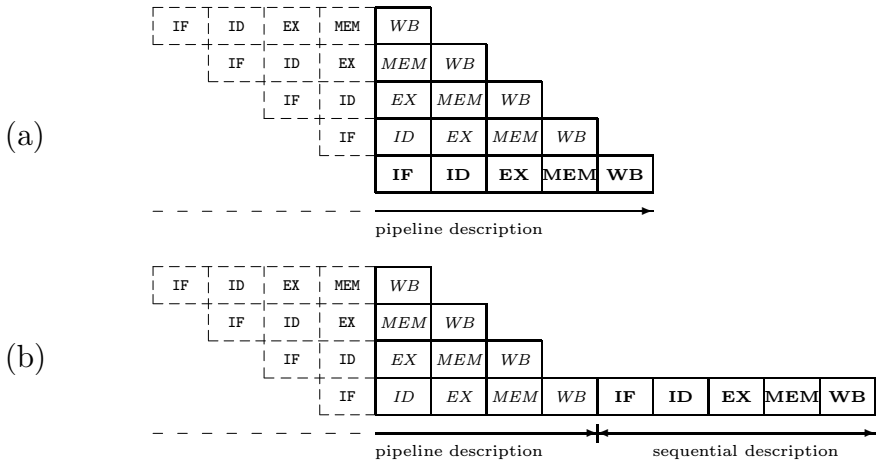


Fig. 4.5: Verification of systems with pipelining

If the two sequences are equivalent then every execution of a program on the system with pipelining can be serialized successively, i.e., each time one more instruction is executed on the sequential processor. Consider the execution of n instructions. In the specification *n-1* pipelined executions are followed by one

serial execution; the implementation consists of n pipelined executions. Both executions produce the same results if the two finite sequences described by the solid areas in Fig. 4.5 are equivalent. By means of an inductive argument, the procedure can then be applied to *n-2* pipelined executions where again one serial execution is extracted. Therefore, an arbitrary program produces the same results on the system with pipelining as on the sequential processor.[5]

Example 4.6

The serialization of the execution of 5 instructions is demonstrated in Fig. 4.6. One instruction is already executed sequentially in Fig. 4.6 (a). A second instruction is executed on the sequential specification in Fig. 4.6 (b). Finally, the entire program of five instructions is executed on the sequential processor in Fig. 4.6 (c). Each of the transformation steps leads to computational equivalent results if the two sequences described by the solid area in Fig. 4.5 are equivalent.

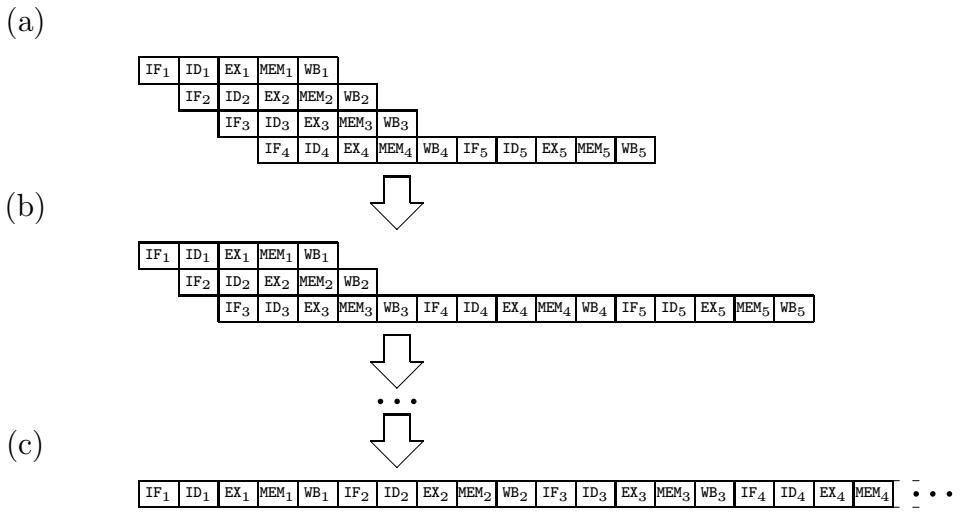


Fig. 4.6: Inductive proof

[BD94] describe the verification process sketched above by transforming an old implementation state in two manners into new specification states which are compared, see also appendix 9.8.

Fig. 4.5 and 4.6 consider only the flushing of a processor without additional stalls due to load-interlocks or branch instructions in the pipeline. The flushing of a 5-stage pipeline may take significantly more than 4 cycles because of those exceptions. Section 7.1 gives results for the verification of pipelined processors which are automatically constructed by a formal synthesis tool developed at Darmstadt University of Technology [Hin00, HRE99]. The generation of the finite sequences according to the technique from [BD94] is completely automatic,

[5]The base case of the induction is to check whether the execution of a single instruction produces the same result on both systems. This case is considered by the equivalence check of the two sequences, too.

see section 7.1. Results for the verification of two *structural* processor descriptions with pipelining are reported in section 7.2.1. The correct flushing of these examples requires some designer information to handle control and data hazards, see section 7.2.1.

Note that pipeline verification according to [BD94] is limited to an equivalence check of the *final* register values which is sufficient, e.g., for general-purpose processor designs. Verification of intermediate results may be also important, e.g., for reactive systems and can be done by our symbolic simulator by simply extending the set of *RegVal*-pairs to be compared.

4.1.4 Expressing the Inherent Timing Structure

The values of the registers after successive assignments are distinguished explicitly by indexing rather than by rewriting the register with the symbolic term assigned to it.

The indexing expresses the inherent timing structure of the initial descriptions explicitly. An indexed register name is called a *RegVal*. A new *RegVal* with an incremented index is introduced after each assignment. An additional upper index s or i distinguishes the *RegVals* of the specification and of the implementation. Only the initial *RegVals* as anchors are identical in the specification and in the implementation, since the equivalence of the two descriptions is tested with regard to arbitrary but identical initial register values. Fig. 4.7 gives a simple

```
adr←pc;
ir←mem(adr);
if ir[0:5]=000111
then (pc←pc+1, adr←ir[6:15]);
     mi←mem(adr);
     ac←ac+mi;
else pc←pc+2;
```

```
adr1 ←pc;
ir1 ←mem(adr1);
if ir1[0:5]=000111
then (pc1 ←pc+1, adr2 ←ir1[6:15]);
     mi1 ←mem(adr2);
     ac1 ←ac+mi1;
else pc1 ←pc+2;
     adr2 ←adr1;
     mi1 ←mi;
     ac1 ←ac;
```

Fig. 4.7: Indexing registers after each new assignment

example written in *LLS*. Parentheses enclose the synchronous parallel transfers in the fourth line. The sequential composition operator ";" separates consecutive transfers.

"Fictive" assignments (italic in Fig. 4.7) have to be generated, if a register is assigned in only one branch of an *if-then-else*-clause in order to guarantee that on each possible path the sequence of indexing is complete and consistent. This makes the indexing complex since nested *if-then-else*-clauses with sequential or parallel assignments have to be considered: the maximum index of all branches has to be determined first; then branches with less assignments have to be filled up correctly with "fictive" assignments.

The number of *RegVals* of a register need not be identical in the specification and in the implementation, see the example given by Fig. 4.8. Therefore, the final *RegVals* are separately marked. Checking computational equivalence consists in

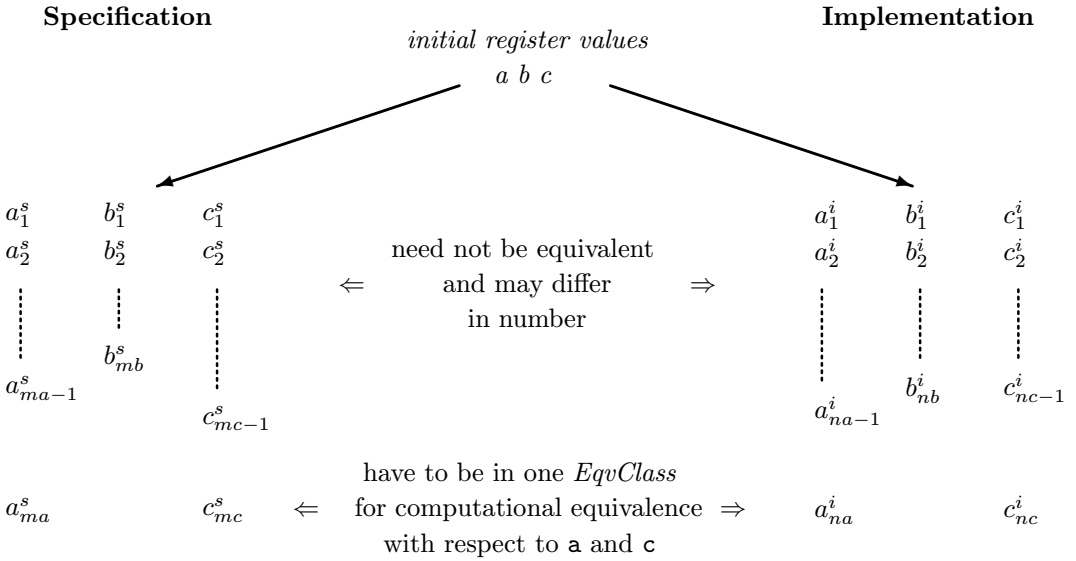


Fig. 4.8: Relation between *RegVals* for computational equivalence

verifying that the final *RegVals* in the specification with the highest index are equivalent to the corresponding final *RegVals* in the implementation on each path, e.g., a^s_{ma} / a^i_{na} and c^s_{mc} / c^i_{nc} in Fig. 4.8.

The introduction of *RegVals* makes all information about the sequential or parallel execution of assignments redundant which is, therefore, removed afterwards.

Formula based techniques like SVC do not use distinct *RegVals* because they represent the modifications of register values in the term-hierarchy implicitly. Expressing the timing structure explicitly has several advantages. Term size explosion is avoided, because terms can be expressed by intermediate *RegVals*. We do not lose information about intermediate relationships by rewriting or canonizing so that arbitrary additional techniques can be used to demonstrate the verification goal. In addition, support of debugging is improved by using the supplementary information.

4.1.5 Memory Operations

The memory model used by the symbolic simulator assumes an unlimited, but finite size for each memory in the descriptions. Similar to [Sho79, BD94, BDL96], two array operations are used to model memory access: `read(mem,adr)` returns the value stored at the address `adr` of memory `mem`. The second operation `store(mem,adr,val)` returns the whole memory state of `mem` after changing the memory state only at `adr` to `val`.

Memories are modeled as vectors (one-dimensional arrays) of words, where a word is in fact a register. We distinguish the two terms for better readability. The words in a memory are numbered with ascending integers starting with 0. Thus $mem[i]$ denotes the $i+1$-th word. Let & denote the concatenation of two words. The j-th *RegVal* of a memory mem is determined by the concatenation of all corresponding words, i.e., $RegVal_j^{mem} = \mathop{\&}_{i=0}^{size(mem)-1} mem_j[i]$. The number of words of the memory is given by $size(mem)$.

`Read-` and `store`-operations are used for all accesses to arrays that are addressed by registers instead of constants. This includes not only, e.g., the data memory of a processor but also the register file. On the other hand, arrays addressed in the descriptions by constants need not be modeled by the `read`/`store`-scheme. A memory word addressed only by a constant can also be considered as a register. This is practically done by replacing all these memory operations by a new distinct register name, e.g., `dmem[3]`←`x` becomes `dmem3`←`x`.

Similar to our procedure for registers, the inherent timing structure of the initial description is expressed explicitly by indexing the memory names. A new *RegVal* (for memories) with an incremented index is introduced after each `store`-operation. For example, the third `store`-operation to a memory `dmem[adr]`←`val` becomes $\texttt{dmem}_3^s \leftarrow \texttt{store}(\texttt{dmem}_2^s, \texttt{adr}_4^s, \texttt{val}_1^s)$. Note that the indexes of `adr` and `val` are arbitrarily chosen in this example. The *RegVals* $\texttt{dmem}_2^s$ and $\texttt{dmem}_3^s$ represent the memory state before and after the `store`-operation. Only the initial register/memory names as anchors are, again, identical in the specification and in the implementation, since the equivalence of the two descriptions is tested with regard to arbitrary but identical initial register values and memory states.

Checking computational equivalence consists in verifying that the state of two memories is identical, i.e., the respective *RegVals* of the memories have to be equivalent. Definition of equivalence requires that $eval(t)$ (see page 12) returns a constant for an acceptable initialization. Definition 2.4 of acceptable initializations has to be modified according to Fig. 4.9 to consider memory operations. $\mathcal{M}$ comprises all memories. The set $\mathcal{R}$ describes all *RegVals* of registers.

$$
\begin{array}{l}
acceptable(init^{RegVals}) \Leftrightarrow \\
\left(\begin{array}{lll} \forall RegVal_{initial,k} \in \mathcal{R} & : & init(RegVal_{initial,k}) \text{ is a constant} \wedge \\ & & init(RegVal_{initial,k}) \in domain(RegVal_{initial,k}) \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall mem \in \mathcal{M} : \forall i = 0, \cdots, size(mem) - 1 : \\ \qquad mem_{initial}[i] \text{ is a constant} \wedge \\ \qquad mem_{initial}[i] \in \textit{domain-of-words}(mem) \end{array} \right) \wedge \\
\left(\begin{array}{lll} \forall C_i \in \mathcal{C} & : & eval(C_i) \textit{ is a constant} \wedge \\ & & \left\{ \begin{array}{lll} C_i \text{ decided true} & : & eval(C_i) = 1 \\ C_i \text{ decided false} & : & eval(C_i) = 0 \end{array} \right. \end{array} \right)
\end{array}
$$

Fig. 4.9: Modification of Definition 2.4 to consider memory operations

The modified definition of an acceptable initialization guarantees only that the words of the initial *RegVal* of a memory are constants. Therefore, defining a `read` as the selection of the corresponding word is only possible if the initial *RegVal* of the memory is read. Furthermore, only the initial *RegVal* of a memory can be evaluated as a concatenation of the corresponding memory words.

Definition 4.1 (`read`- and `store`-operations)

$$RegVal^{mem}_{initial} \quad : \quad \mathop{\&}_{i=0}^{size(mem)-1} mem_{initial}[i]$$

$$\texttt{read}(RegVal^{mem}_{initial}, adr) \quad : \quad mem_{initial}[adr]$$

$$\texttt{read}(RegVal^{mem}_{j\neq initial}, adr) \quad : \quad t = \begin{cases} RegVal^{mem}_{j-1} \ : \ \texttt{read}(RegVal^{mem}_{j-1}, adr) \\ store(RegVal^{mem}_{j-1}, sadr, val) \ : \\ \qquad \textbf{if } adr = sadr \\ \qquad\quad \textbf{then } val \\ \qquad\quad \textbf{else } \ \texttt{read}(RegVal^{mem}_{j-1}, adr) \end{cases}$$

t : *right-hand side term of assignment to* $RegVal^{mem}_{j}$

$$\texttt{store}(RegVal^{mem}_{j}, adr, val) \ : \ \left(\mathop{\&}_{i=0}^{adr-1} \texttt{read}(RegVal^{mem}_{j}, i)\right) \ \& \ val \ \& \ \left(\mathop{\&}_{i=adr+1}^{size(mem)-1} \texttt{read}(RegVal^{mem}_{j}, i)\right)$$

The definition of `read`- *and* `store`*-operations supposes that only (preceding) RegVals of the same memory or* `stores` *are assigned to RegVals of memories.*

If the `read`-operation accesses an initial memory state then the corresponding initialization of the data word $mem_{initial}[adr]$ of memory *mem* is returned. Otherwise the `read`-operation is applied to the last preceding `store`-operation. If the values of the addresses are the same then the corresponding value stored is read. Otherwise it seems for the `read` that the preceding `store` was not executed and the value at the same address is read from the previous memory state.

The value of a `store`-operation, which returns the entire new memory state, is defined as a concatenation of `read`-operations of all words, considering the new value *val* at *adr*. The value of *RegVals* of memories is defined by the `store`-operation or the *RegVal* assigned, see Definition 2.3 of $eval(t)$. Two memory states are identical iff all data words are identical. As in Definition 2.6, two terms are intuitively equivalent if an exhaustive numerical simulation of each possible initialization of the registers and memories result in the same value for both terms.

The assumption of an arbitrary memory size requires verifying that the address is not out of range of the actual memory. This is trivial in most of the cases, where memory size is $size(mem) = 2^{addresslines}$.

Note that addresses and values in Fig. 4.9 are *constants* while the equivalence detection for memory operations described in section 5.9 has to cope with *symbolic* addresses.

4.2 Invoking the Equivalence Detection

The symbolic simulator employs a number of techniques to determine equivalent terms during simulation. Re-checking equivalence for all terms already encountered on a path after each simulation step would decrease the simulation speed unacceptably. Therefore, invoking the equivalence detection has to be controlled as discussed in this section. The *dd-checks* are usually just used at the end of a path if the verification goal is not demonstrated. An exception represents symbolic simulation for gate-level verification as discussed in section 6.4.

The transformation steps done during pre-processing preserve the timing structure. In general, equivalence of the arguments of two terms is already known, when the second term is found on the path. Therefore, it is sufficient to check only at the first occurrence of a term whether it is equivalent to other terms previously found. Furthermore, equivalence checking for a term is stopped after the first union operation, since all equivalent terms are (ideally) already in the same equivalence class.

Invoking equivalence detection for a term only at its first occurrence can be insufficient because of successive case-splits. The set of possible initial *RegVals* is constrained by a case-split. Equivalence of two terms previously found on the path might be given only under this new decision.

Example 4.7
The last situation occurs especially in the case of operations to memories. The order of the `read`*- and the* `store`*-operation is reversed in the implementation of the example of Fig. 4.10. Thus,* `val` *is forwarded if the addresses are identical. The problem is to detect that, in the opposite case, the final values of* `x` *are identical, which is only obvious after the case-split (setting adr1* $\not\cong_C$ *adr2) and not already after the assignments to* `x`*.*

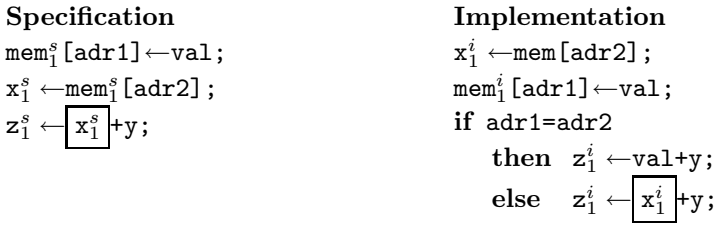


Fig. 4.10: Forwarding example

The example indicates, that it is important to check `read-` *and* `store`*-terms whenever the equivalence classes of the corresponding addresses are modified.*

Re-checking equivalence of all terms found on a path after each case-split is unacceptable, too. Equivalence detection is invoked again for a term in two cases:

- the value of a condition cannot be decided, i.e., its value seems to depend on the initial *RegVals*. This would make a case-split necessary. The terms of the condition are re-checked if there are additional case-splits *after* the first occurrence of the terms. The repeated equivalence check verifies if additional equivalences are given under the additional assumptions of the case-splits. Those equivalences may allow to decide the value of the condition and to avoid the case-split leading to one false path;

- the verification goal, i.e., the equivalence of two terms or *RegVals* is not demonstrated since the terms are not in the same *EqvClass*.

Terms can have other terms, intermediate *RegVals* and initial *RegVals* as arguments. Invoking the equivalence detection for the *arguments* of a term, i.e., the subterms depends on whether the term is found for the first time or whether the equivalence of the term is re-checked:

- *a term is found for the first time on a path:* equivalence detection is called recursively only for those subterms, which have also been found for the first time; note that the terms assigned to intermediate *RegVals* are guaranteed to be checked at least once;

- *equivalence of a term is re-checked:* all arguments are re-checked recursively; terms assigned to intermediate *RegVals* are re-checked, too. Therefore, invoking recursively the equivalence detection stops only at the initial *RegVals* or constants.

Invoking the equivalence detection only when a term is first found, a condition has to be decided, or the verification goal is not demonstrated need not be optimal. Invoking additionally the equivalence detection after case-splits can be useful if a term is frequently used as argument of other terms and

- if the equivalence of a term with a specific function to other terms often depends on *successive* case-splits,

- it is frequent that the assumption of a case-split establishes an equivalence between one of the terms or subterms of the condition and some other term, or/and

- the additional equivalence check requires little computation time.

Deciding if an additional check is useful is a trade-off between its computation time and the time for a *possible* re-check, which is often higher. If the equivalence of two terms has to be detected to decide a condition or to demonstrate the verification goal then a re-check is required as described above. This re-check considers all subterms and requires, therefore, more computation time. For example, a re-check of the final values of $\mathtt{z}_1^s$ and $\mathtt{z}_1^i$ in Example 4.7 includes re-checking the additions. This is avoided if equivalence detection is invoked again for the `read`-operation $\mathtt{mem}_1^s\mathtt{[adr2]}$ directly after the case-split.

The effect of invoking additionally the equivalence detection on the simulation speed has to be judged by experimental evidence. The following additional checks have turned out to be useful:

- memory operations are re-checked each time the *EqvClass* of the corresponding addresses is modified. This is necessary since the value of the addresses is often constrained by case-splits *after* the first occurrence of the term as in Example 4.7;

- a case-split can constrain the value of a term so that the term is equivalent to a constant; since the domain of an n-bit-vector is restricted to 2^n values, setting it $\not\cong_{\mathcal{C}}$ to $2^n - 1$ values means that it must be equivalent to the remaining value. For example, if `b`, a vector of 2 bits, is set inequivalent to `00`, `01`, and `11`, then `b` is equivalent to `10`. Moreover, setting *bit-selections* of a term equivalent to a constant (e.g., $\mathtt{a[3:4]} \cong_{\mathcal{C}} \mathtt{3}$) in a case-split constrains also the set of possible values of a term. Therefore, the technique described in section 5.10 is used to check whether a term is equivalent to a constant each time

  - the term is set inequivalent to a term, which is in a *EqvClass* with a constant,
  - a *bit-selection* of the term is set equivalent to a constant, or
  - a *bit-selection* of the term is set inequivalent to a term, which is in an *EqvClass* with a constant.

  Invoking equivalence detection in these cases is useful since knowledge about constant values of terms often simplifies significantly equivalence detection;

- the result of each *dd-check* is marked since it might be reused during the simulation of the remaining paths. If the conditions under which the previous *dd-check* was performed are also satisfied in the current path then the equivalence verified by the *dd-check* holds, too; section 6.6 describes how results of *dd-checks* are notified and when the conditions are checked.

4.3 Notifying Results at Equivalence Classes

EqvClasses permit to notify the results of the symbolic simulation. Equivalent terms are collected in *EqvClasses*. Therefore, checking whether two terms are equivalent consists of comparing their *EqvClass*. Furthermore, inequivalences are notified at the *EqvClasses*. If two terms are identified to be inequivalent then the inequivalence is marked at both corresponding *EqvClasses*. All other terms of the two *EqvClasses* are marked in this way as inequivalent, too.

Notifying the inequivalence of *EqvClasses* with constants is not necessary since two *EqvClasses* with constants are in any case inequivalent. Including the constant in the list of members of the *EqvClass* is not efficient. It is frequently tested during symbolic simulation if an *EqvClass* contains a constant. These tests would make it necessary to go through the list of members. Therefore, constants are separately marked at *EqvClasses*.

EqvClasses are created initially only for those constants which appear explicitly in the descriptions being compared. The dynamic creation of *EqvClasses* during the symbolic simulation can become necessary if the equivalence detection detects the equivalence of a term to a constant which does not appear explicitly.

Example 4.8
A description contains the clause **if** `a=7` **then** `x←a[1:0]` .... *The EqvClass for the constant 7 is created during pre-processing. The terms* `x` *and* `a[1:0]` *in the* **then***-branch are equivalent to the constant 3. It is detected during symbolic simulation that an EqvClass with this constant has to be created if the constant does not appear explicitly elsewhere in the description.*[6]

Constants, which are described as bit-vectors in *LLS/IDS*, are translated to integers during pre-processing, e.g., `(CONST 1 1 0)` is transformed to 6. Avoiding the representation as a bit-vector reduces the size of the descriptions and permits a significantly faster comparison of constants during symbolic simulation.[7]

The unification of two *EqvClasses* is implemented as the elimination of one of the *EqvClasses*. The unification procedure guarantees that an *EqvClass* with a constant is never eliminated.[8] The remaining *EqvClass* inherits from the eliminated *EqvClass*:

- the members;

[6]The creation of an *EqvClass* can be avoided by assigning the new constant to the *EqvClass* of the terms `x` and `a[1:0]`. This approach is avoided since it violates the separation of equivalence detection and unification of *EqvClasses* in the implementation of the simulation tool.

[7]The length of the initial bit-vector need not be notified: a constant is either compared or assigned to a term or a *RegVal*; their length is available during symbolic simulation. Compatibility of the bit-vector length is checked during pre-processing.

[8]Two *EqvClasses* with constants are never unified.

- the list of inequivalent *EqvClasses*; it is not necessary to consider *EqvClasses* with a constant in this list if the remaining *EqvClass* contains a constant;

- the list of `read`-operations, which use one of the terms in the *EqvClass* as address, see section 5.9 and 4.2;

- restrictions concerning the range of the terms in the *EqvClass*. For example, if `x<5` is decided to be true in a case-split, then the *EqvClass* of `x` has a restriction "$<$ `5`"; Section 5.5 discusses how the information about these restrictions is used to detect equivalences and to decide conditions consistently;

- the list describing which bits of the terms in the *EqvClass* are identified to be equivalent to constants; this information is obtained basically if there is a concatenation term in the *EqvClass*; if one of the arguments of the concatenation is equivalent to a constant then the corresponding subvector of the concatenation term is notified as constant.[9] For example, the term `x[2:0] & y[6:0]` is constant at the bit positions 8 to 10 if `x` is equivalent to a constant. The unification with another *EqvClass* can reveal that all bits are equivalent to constants; another unification with the *EqvClass* of the resulting constant follows in this case.

After inheriting the properties of the eliminated *EqvClass* it is checked if one of the results of a previous *dd-check* can be reused, see section 4.2 and 6.6. Furthermore, `read`- or `store`-operations with addresses in the *EqvClass* are rechecked, see section 4.2 and 5.9.

Note that terms in the same *EqvClass* need *not* have the same bitvector-length.

Example 4.9
The terms `a[2:0]` *and* `b[1:0]` *are in the same EqvClass, if they are both equivalent to the same constant. The same holds for the concatenation* `000&a[4:0]` *and the subterm* `a[4:0]` *although the length of the first term is greater.*

This fact is considered in the *dd-checks* described in chapter 6 when substituting a term by another term in the same *EqvClass* during formula construction.

Practically, the union-operation of two *EqvClasses* caused by an assignment is very simple. The *EqvClass* of the *RegVal* on the left-hand side of the assignment is guaranteed to be unmodified. Therefore, it is sufficient to change the *EqvClass* of the *RegVal* and to mark it as an additional member of the *EqvClass* of the assigned term.

[9] This is redundant, if each subterm is equivalent to a constant; the concatenation is in the *EqvClass* of the resulting constant in this case.

4.4 Accelerating the Decision Procedure by *CondBits*

Symbolic simulation requires a decision algorithm each time an *if-then-else*-clause is reached. The condition has to be evaluated in order to determine whether a case-split is required on the current path or not. Identifying *CondBits* in the conditions accelerates this decision procedure. *CondBits* replace

(a) tests for equality of bit-vectors, i.e., terms or *RegVals* (e.g., $\mathtt{r}_3^s = \mathtt{x}_2^s \ \mathtt{+}\ \mathtt{y}_1^s$);

(b) all terms with Boolean result (e.g., $\mathtt{r}_3^s \ \mathtt{<}\ \mathtt{x}_2^s$) except the connectives below;

(c) single-bit registers (e.g., status-flags).

After the replacement, the conditions of the *if-then-else*-clauses contain only *condition terms* and *CondBits*. A *condition term* consists of one of the propositional connectives (`not`, `nand`, `nor`, `and`, `or`, `xor`)[10] and a list of *CondBits* and/or other *condition terms*. Identical comparisons might be done multiple times on one path. Multiple evaluation of the same condition is avoided by assigning one of three values (*undefined*, *true*, *false*) to the *CondBits*. If a *CondBit* appears for the first time on a path, its value is *undefined*. Therefore, its condition is checked by comparing the equivalence classes of two terms or *RegVals*: In case (a), we have to check the terms on the left-hand and right-hand side, whereas in cases (b) and (c) the equivalence class of the term is compared to the equivalence class of the constant 1. There are three possible results:

i. the two terms to be compared are in the same equivalence class. The *CondBit* is asserted or *true on this path* for any acceptable initialization of the registers and memories;

ii. the equivalence classes of the terms are inequivalent or contain different constants. The *CondBit* is in any case denied or *false*;

iii. otherwise the *CondBit* may be true or false, depending on the initial register and memory values. Both cases have to be examined in a case-split. Denying/asserting a *CondBit* leads to a decided inequivalence or union-operation.

The inconsistency check in the symbolic simulation algorithm of section 2.8 (line 6 in Algorithm 2.1) and 4.6 (line 19 in Algorithm 4.1) determines if the condition of a *CondBit* has been decided inconsistently. The incomplete equivalence detection during symbolic simulation can cause such inconsistent decisions. If the equivalence or inequivalence of the two terms compared has not been detected then a case-split follows erroneously. One of the cases leads to a false path.

[10] A check for equality is replaced by a *CondBit*.

The condition of an *if-then-else*-clause is either a *CondBit* or a *condition term* (see above) which has itself *CondBits* or other *condition terms* as arguments. Its value is determined in a depth first search. The value of more than one *CondBit* of a *condition term* might depend on the initial register values.[11] The first *CondBit* found with unfixed value is set as candidate for the next case-split. However, the other arguments of the *condition term* - which might be *CondBits* or other *condition terms* - are still evaluated since they might determine the value of the *condition term*.

Example 4.10
Fig. 4.11 gives an example for the evaluation of a condition in our internal prefix notation.

(and (nand $CondBit_2$ $CondBit_3$ $CondBit_5$) $CondBit_1$
(nor $CondBit_2$ $CondBit_4$))

CondBit	Value on current path
$CondBit_1, CondBit_2, CondBit_3$	depends on initial *RegVals*
$CondBit_4$	*true*
$CondBit_5$	*false*

Fig. 4.11: Example for the evaluation of conditions

The arguments of the nand*-term are evaluated first.* $CondBit_2$ *is noted as first candidate for the next case-split since its value depends on the initial RegVals. But the value of* $CondBit_5$ *is false, i.e., the value of the* nand*-term is determined to be true. Therefore, the* nand*-term does not require a case-split and the candidate is cleared.*

$CondBit_1$ *is set as new candidate next since its value depends on the initial RegVals. The same holds for the first argument* $CondBit_2$ *of the* nor*-term. The candidate remains unchanged. The value of the* nor*-term is determined next by the second argument* $CondBit_4$ *to be false independently of the value of* $CondBit_2$*. Therefore, the value of the* and*-term is determined, too. The candidate for the next case-split is cleared and no case-split is performed.*

Evaluation of the arguments, i.e., the *CondBits* is stopped, if the value of the *condition term* is determined. For example, $CondBit_3$ and $CondBit_5$ of the nand-term in Fig. 4.11 are not evaluated if the value of $CondBit_2$ is *false*.

4.5 Examples of Symbolic Simulation Runs

Two examples are given in the following to illustrate the progress of a symbolic simulation:

- the parallel simulation of a single path of the example in Fig. 2.2 comparing two rtl-descriptions, and

[11] Except for the propositional connective "not".

- the simulation of the example in Fig. 2.3 comparing a rtl- and a gate-level description. This simulation is *not* performed in parallel, see below.

4.5.1 RTL against RTL

Fig. 4.12 (a) shows the example of Fig. 2.2 after pre-processing (see section 4.1). The symbolic simulation of one path during the equivalence check of the example is described in Fig. 4.12 (b). The members of the *EqvClasses* after every simulation step are given. Initially, all terms and *RegVals* are in distinct *EqvClasses*. S1 is simulated first. When symbolic simulation reaches S2, the

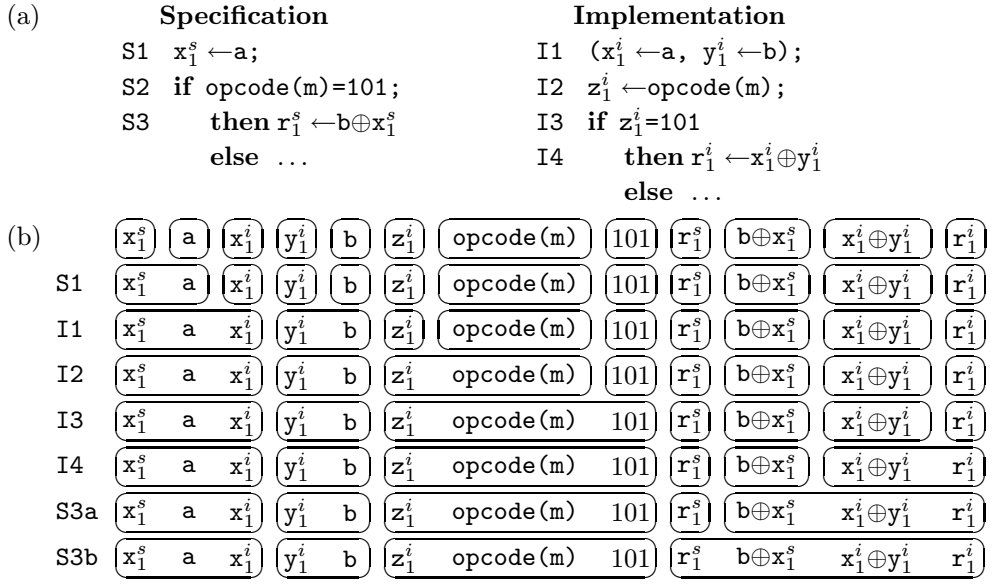


Fig. 4.12: Simulation run of two descriptions at rt-level

condition of S2 depends on the initial *RegVals* (case iii on page 53) and the simulation is blocked. Paths are searched simultaneously in specification and implementation. After the simulation of I1 and I2, I3 requires also a case-split. Decisions in the normally more complex implementation have priority in order to facilitate a parallel progress. Therefore, a case-split on the condition in I3 is performed. Only the case with the condition asserted is sketched in Fig. 4.12, where the equivalence classes of z_1^i and the constant 101 are unified and I4 is simulated. The condition of S2 is now decidable in the given context since both sides of the condition are in the same *EqvClass* (case i on page 53), i.e., no additional case-split is required. First the equivalence of $b \oplus x_1^s$ and $x_1^i \oplus y_1^i$ is detected (S3a) and then the assignment to r_1^s is considered (S3b). Finally, r_1^s and r_1^i are in the same equivalence class. Therefore, computational equivalence is satisfied at the end of this path. Equivalence would be denied if they were in different equivalence classes. Note that simultaneous progress in implementation and specification avoids simulating S1 again for the else-case.

4.5.2 RTL against Gate-level

Parallel simulation as described in the previous example is not reasonable when comparing a rt- and a gate-level description. The gate-level simulation typically does not require any additional case-splits, i.e., the selection of the relevant path is mainly determined by the case-splits during the simulation of the specification at rt-level. A parallel simulation would lead to an entire simulation of the implementation *without* the information of the case-splits since the simulation of the specification is blocked at the first case-split. Only few equivalences are detected at gate-level if no specific path has been taken. Therefore, a complete path is first simulated in the specification. The information obtained from this path is used to detect equivalences during the following simulation of the implementation.

Fig. 4.13 gives the two sequences to be compared for the verification of the example in Fig. 2.3. The (structural) implementation is duplicated since two cycles have to be simulated. The assignment to the register `r` is modeled as a concatenation of the gate-level expressions at the corresponding flip-flop inputs. The single bits (e.g., $\mathtt{r}_1^i\mathtt{[0]}$) do not occur explicitly in the sequences to be simulated. However, equivalences of those single bits and other expressions are also detected and noted during symbolic simulation as if those selections occurred explicitly. Note that the bits of the registers are equivalent to the corresponding expressions in Fig. 4.13, e.g., $\mathtt{r}_1^i\mathtt{[0]} \cong_{\mathcal{C}} \mathtt{(ctrl}_1^i\ \mathtt{nand\ m)\ and\ (not\ r[0])}$.

The specification is simulated first. The *EqvClasses* of $\mathtt{r}_1^s$ and `r+1` are unified (first line). The condition of the specification depends on the initial value of `m`, i.e., a case-split follows. The `then`-path is reached in the first case with the assumption `m=0`. Finally, the *EqvClasses* of $\mathtt{r}_2^s$ and $\mathtt{r}_1^s + 1$ are unified.

Specification	**Implementation**
$\mathtt{r}_1^s \leftarrow$`r+1;`	$\mathtt{ctrl}_1^i \leftarrow$`0` ;; *assumption about initialization*
if `m=0`	;; *first cycle*
then $\mathtt{r}_2^s \leftarrow \mathtt{r}_1^s$`+1;`	$\mathtt{r}_1^i \leftarrow$`(ctrl`$_1^i$ `nand m) and (r[2] xor (r[1] and r[0])) &`
else $\mathtt{r}_2^s \leftarrow$`"000";`	`(ctrl`$_1^i$ `nand m) and (r[1] xor r[0]) &`
	`(ctrl`$_1^i$ `nand m) and (not r[0])`
	$\mathtt{ctrl}_2^i \leftarrow$`not(ctrl`$_1^i$`)`
	;; *second cycle*
	$\mathtt{r}_2^i \leftarrow$`(ctrl`$_2^i$ `nand m) and (`$\mathtt{r}_1^i$`[2] xor (`$\mathtt{r}_1^i$`[1] and `$\mathtt{r}_1^i$`[0])) &`
	`(ctrl`$_2^i$ `nand m) and (`$\mathtt{r}_1^i$`[1] xor `$\mathtt{r}_1^i$`[0]) &`
	`(ctrl`$_2^i$ `nand m) and (not `$\mathtt{r}_1^i$`[0])`
	$\mathtt{ctrl}_3^i \leftarrow$`not(ctrl`$_2^i$`)`

Fig. 4.13: Descriptions to simulate for the verification of the example in Fig. 2.3

The least significant bit in the assignment to $\mathtt{r}_1^i$ is examined first in the implementation. The following equivalences are detected and the corresponding *EqvClasses* are unified if *no* intermediate *dd-checks* are performed (see below):

- $\texttt{ctrl}_1^i \cong_{\mathcal{C}} 0$ which is the assumption about the initialization of `ctrl`, see section 2.3;

First cycle:

- $\texttt{(ctrl}_1^i \texttt{ nand m)} \cong_{\mathcal{C}} 1$ because of the initialization of $\texttt{ctrl}_1^i$;
- $\texttt{r}_1^i\texttt{[0]} \cong_{\mathcal{C}} \texttt{(not r[0])}$ since the first argument of the `and`-term is 1;
- $\texttt{r}_1^i\texttt{[1]} \cong_{\mathcal{C}} \texttt{r[1] xor r[0]}$; note that the term $\texttt{(ctrl}_1^i \texttt{ nand m)}$ is *not* evaluated again during the examination of the two most significant bits of $\texttt{r}_1^i$, see section 4.4;
- $\texttt{r}_1^i\texttt{[2]} \cong_{\mathcal{C}} \texttt{r[2] xor (r[1] and r[0])}$;
- $\texttt{r}_1^i$ and the concatenation of the individual bits of $\texttt{r}_1^i$; no equivalence is detected for the concatenation;
- $\texttt{not(ctrl}_1^i\texttt{)} \cong_{\mathcal{C}} 1 \cong_{\mathcal{C}} \texttt{ctrl}_2^i$;

Second cycle:

- $\texttt{ctrl}_2^i \texttt{ nand m} \cong_{\mathcal{C}} 1$ since `m` is decided to be equivalent to 0 in this case;
- $\texttt{r}_2^i\texttt{[0]} \cong_{\mathcal{C}} \texttt{(not r}_1^i\texttt{[0])}$ since the first argument of the `and`-term is 1; moreover, the *EqvClasses* of $\texttt{r}_2^i\texttt{[0]}$ and `r[0]` are unified;
- $\texttt{r}_2^i\texttt{[1]} \cong_{\mathcal{C}} \texttt{r}_1^i\texttt{[1] xor r}_1^i\texttt{[0]}$; the term $\texttt{(ctrl}_2^i \texttt{ nand m)}$ is not evaluated again;
- $\texttt{r}_2^i\texttt{[2]} \cong_{\mathcal{C}} \texttt{r}_1^i\texttt{[2] xor (r}_1^i\texttt{[1] and r}_1^i\texttt{[0])}$;
- $\texttt{r}_2^i$ and the concatenation of the individual bits of $\texttt{r}_2^i$; no equivalence can be detected without *dd-check* for the concatenation;
- $\texttt{not(ctrl}_2^i\texttt{)} \cong_{\mathcal{C}} 0 \cong_{\mathcal{C}} \texttt{ctrl}_3^i$.

Finally, the terms $\texttt{r}_2^s$ and $\texttt{r}_2^i$ are *not* in the same *EqvClass*, i.e., computational equivalence is not demonstrated. Therefore, the more powerful *dd-checks* are used to compare the final values of `r` in both descriptions. The results obtained during symbolic simulation are used to simplify the *dd-check*. A simple backward-substitution without using the information of the *EqvClasses* would require the construction of the decision diagrams for the expression in Fig. 4.14 (a). Fig. 4.14 (b) shows the expression which is verified using decision diagrams in our symbolic simulator without intermediate *dd-checks* (see below). Note that the benefit of using results of the other equivalence detection techniques increases significantly if the number of sequential steps is higher and the Boolean expressions in each step are more complex than in our simple example.

(a) without using information of *EqvClasses*

```
r+1+1≡ (not(ctrl) nand m) and
           (((ctrl nand m) and (r[2] xor (r[1] and r[0]))) xor
            (((ctrl nand m) and (r[1] xor r[0])) and
             ((ctrl nand m) and (not r[0])))) &
       (not(ctrl) nand m) and
           (((ctrl nand m) and (r[1] xor r[0])) xor
            ((ctrl nand m) and (not r[0]))) &
       (not(ctrl) nand m) and ((ctrl nand m) and (not r[0]))
```

(b) using information of *EqvClasses*

```
r+1+1≡ ((r[2] xor (r[1] and r[0])) xor ((r[1] xor r[0]) and (not r[0]))) &
       ((r[1] xor r[0]) xor (not r[0])) &
       r[0];
```

(c) using additional intermediate *dd-checks*

```
r+1≡   (r[2] xor (r[1] and r[0]))&
       (r[1] xor r[0]) &
       not(r[0]);
```

Fig. 4.14: Expressions to verify by *OBDDs* with and without considering simulation results

No *dd-check* is required for the case `m` $\not\cong_{\mathcal{C}}$ `0`. The `else`-branch in the specification is reached and the *EqvClass* of $\mathtt{r}_2^s$ and 0 are unified. The following equivalences are detected and remarked in the implementation:

- *steps of first cycle identical to the list on page 56*;
- `(ctrl`$_2^i$ `nand m)` $\cong_{\mathcal{C}}$ 0 since $\mathtt{ctrl}_2^i$ and `m` are both equivalent to 1;
- all three bits $\mathtt{r}_2^i$`[0]` to $\mathtt{r}_2^i$`[2]` are identified to be equivalent to 0;
- during the examination of the concatenation assigned to $\mathtt{r}_2^i$, first $\mathtt{r}_2^i$`[1:0]` $\cong_{\mathcal{C}}$ 0 and then $\mathtt{r}_2^i \cong_{\mathcal{C}} 0$ is detected.

Finally, $\mathtt{r}_2^s$ and $\mathtt{r}_2^i$ are both in the same *EqvClass* and computational equivalence is demonstrated in this path without additional *dd-check*.

The formula to be checked by decision diagrams in the first path is even simpler if *intermediate dd-checks* are applied, see Fig. 4.14 (c). These checks can be used *during* the path search if no equivalence has been found yet for a term assigned to a *RegVal* at gate-level. This is the case for the terms assigned to $\mathtt{r}_1^i$ and $\mathtt{r}_2^i$. The first intermediate *dd-check* reveals the equivalence of $\mathtt{r}_1^i$ and $\mathtt{r}_1^s$ by checking the formula in Fig. 4.14 (c). The second *dd-check* uses these two equivalent terms as *dd-cutpoints*, i.e., $\mathtt{r}_1^i$ and $\mathtt{r}_1^s$ are considered as if they were "primary inputs", see section 6.2. Therefore, the same formula is established as in Fig. 4.14 (c) for the first *dd-check*, only the cutpoint for $\mathtt{r}_1^{s/i}$ is used instead of `r`. The second formula is *not* checked by *OBDDs*, since the similarity to the first formula is detected and the previous result is reused, see section 6.2. The same holds for simulation of the second path with `m` $\not\cong_{\mathcal{C}}$ `0`. Intermediate *dd-checks* are motivated in section 4.6 and described with examples in section 6.4.

Detecting the equivalence of the datapath-operation "+" and the corresponding gate-level expression requires a more time-consuming *dd-check* during the simulation of the first path for $\mathtt{m} \cong_{\mathcal{C}} 0$. Normally such datapath-operations are performed on separate blocks, e.g., an adder-block from a standard library. Those standard blocks are replaced for symbolic simulation during the pre-processing by the corresponding high-level operation (e.g., "+"), see appendix 9.4. First, this replacement avoids using the more time-consuming *dd-checks* during symbolic simulation. Second, the standard blocks can be verified separately against their high-level specification by combinatorial equivalence checking. The gate-level expressions of the incrementer in the implementation are used in the example of Fig. 4.13 *only* to give a first impression about the use of *dd-checks*. More elaborated examples requiring *dd-checks* are presented in chapter 6.

(a)

```
ctrl1^i ← 0      ;; assumption about initialization
if (ctrl1^i nand m)
  then r1^i ← inc(r)[2] & inc(r)[1] & inc(r)[0];
  else  r1^i ←"000";
ctrl2^i ← not(ctrl1^i)
if (ctrl2^i nand m)
  then r2^i ← inc(r1^i)[2] & inc(r1^i)[1] & inc(r1^i)[0];
  else  r2^i ←"000";
ctrl3^i ← not(ctrl2^i)
```

(b)

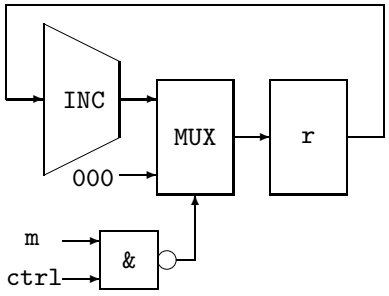


Fig. 4.15: Replacing standard blocks by high-level operations

Fig. 4.15 (a) gives the sequence to simulate if the standard incrementer block is not broken into gates in contrast to Fig. 4.13. This block is replaced instead by the datapath-operation "inc" for symbolic simulation in Fig. 4.15 (a). Note that the standard blocks in the output description of the synthesis tool are easily identified since they are described as separate components as in Fig. 4.15 (b). No *dd-check* is required to demonstrate equivalence of the sequence in Fig. 4.15 (a) and the specification in Fig. 4.13.

4.6 Implementation of the Symbolic Simulation Algorithm

The recursive symbolic simulation algorithm presented in section 2.8 is modified for optimization. The implemented version is given by Algorithm 4.1. The modifications necessary for verification at gate-level are described below.

Lines 3 to 10 of Algorithm 4.1 summarize the path search. The specification and the implementation are simulated in parallel. A case-split is performed when

Algorithm 4.1 Implemented symbolic simulation

```
INPUT spec, impl;
 1. push (dummy_cond,spec,impl) rem_cases;
 2. while rem_cases ≠ ∅ do
 3.       act_case := pop(rem_cases);
 4.       assert(act_case_to_decide);
 5.       repeat
 6.         to_decide:= { - simulate act_case_spec and act_case_impl
                          in parallel until next condition
                          depending on initial RegVals
                        - reduce act_case_spec/impl accordingly
                        - return condition to decide }
 7.         if to_decide is found then
 8.             push (to_decide,act_case_spec,act_case_impl) rem_cases;
 9.             deny(to_decide);
10.       until to_decide not found;
11.       if ∃k : NOT ( R^spec_final,k ≅_C R^impl_final,k ) then
12.          check_additional_properties;
13.          recheck_equivalence_of_terms;
14.          if ∃k : NOT ( R^spec_final,k ≅_C R^impl_final,k ) then
15.             ∀k : NOT ( R^spec_final,k ≅_C R^impl_final,k ) : LET F_k ⇒ R^spec_final,k ≅_C R^impl_final,k;
16.             if ∃k : F_k ≡ TRUE then
17.                   mark_new_relations_found;
18.                   pop(rem_cases) until a term in F_k has not appeared;
19.             elsif ∃C_i ∈ C:  inconsistent(C_i) then
20.                   mark_new_relations_found;
21.                   pop(rem_cases) until inconsistent decision reached;
22.             else report_debug_information;
23.                   return(FALSE);
24. od;
25. return(TRUE);
```

simulation reaches a condition `to_decide` that cannot be decided in general but depends on the initial register values (line 6 and 10). For every case-split due to a condition `to_decide`, first the denied case is examined (line 9) while the asserted case is stored on the stack `rem_cases` (line 8). Each element of `rem_cases` is a triple ($\texttt{act\_case}_{to\_decide}$, $\texttt{act\_case}_{spec}$, $\texttt{act\_case}_{impl}$). $\texttt{act\_case}_{to\_decide}$ denotes

the condition of the case-split and $\texttt{act\_case}_{spec/impl}$ describe the remaining parts of specification/implementation to be simulated after the case-split. Initially, `rem_cases` contains as single element the whole specification and implementation with a "dummy"-condition (line 1).[12] Note that only those parts of the descriptions that are not yet simulated in this path are examined after case-splits, since $\texttt{act\_case}_{spec/impl}$ are reduced during simulation (line 6).

A complete path is found when no more condition `to_decide` is found and the end of both descriptions is reached. The computational equivalence of the descriptions in this path is tested by checking whether the relevant final *RegVals* $R^{spec/impl}_{final,k}$ are in the same *EqvClass* (line 11).

Lines 12 to 23 describe the case where computational equivalence is not reported at the end of a path. If the verification goal is not given in a path, then the first step is to consider additional function properties which are less often useful to consider or more time consuming to check (line 12). Moreover, equivalence detection is invoked again for all terms assigned to the final *RegVals* (line 13). This check is recursive and terms assigned to intermediate *RegVals* are re-checked, too, see section 4.2. Invoking recursively the equivalence detection stops only at the initial *RegVals* or constants.

If the verification goal is not yet reported for all pairs of final *RegVals* an attempt is made to decide the equivalence by performing *dd-checks* (lines 15 to 21). The *dd-checks* are described in detail in chapter 6. Formulas are built considering knowledge about path-dependent equivalence/inequivalence of intervening terms. These formulas are sufficient for the equivalence of the final *RegVals* (line 15). A pre-check follows, which applies some logic minimization techniques and which checks whether a formula was built previously and stored in a hash-table. New formulas are checked using binary decision diagrams. This is the first time a canonical form is built.

If none of the formulas is satisfiable, then all decided *CondBits*, i.e., conditions for which a case-split was done, are checked in order of their appearance. A formula for the value of the condition is built and verified using *OBDDs*, too. This check has to reveal if a contradictory decision due to the incomplete equivalence detection on the fly led to a false path. Using the information of the *EqvClasses* again facilitates considerably building the required formulas.

The path is backtracked if at least one formula is valid (line 16) or if a contradictory decision has been detected (line 19). Backtracking is done by popping elements from the stack `rem_cases`. Each time, the corresponding context is restored. Backtracking is stopped if

- (*case line 18*) at least one of the terms appearing in a valid formula F_k has not appeared yet on the path;

[12] The "dummy"-condition is only used to complete the triple. Asserting this "dummy"-condition in line 4 has no effect.

- (*case line 21 and C_i asserted*) the value of the condition C_i, which has been decided inconsistently, is *undefined* in the current context. All succeeding case-splits are due to the inconsistent decision (C_i is *true*) and need not be considered; note that the case with the consistent decision (C_i is *false*) has been already checked;

- (*case line 21 and C_i denied*) the condition $\texttt{act\_case}_{to\_decide}$ of the top element on the stack `rem_cases` is C_i; simulation continues with this stack; the (consistent) asserted case is verified by popping the top element from `rem_cases` in line 3.

The new detected relationship is marked before backtracking so that it is checked during further path search on the fly (line 17 and 20). Probability is high that also on other paths the more time consuming algorithms are invoked unnecessarily again due to this relationship. Furthermore, deciding one more time the same condition inconsistently is avoided.

Finally, computational equivalence is denied and the counterexample is reported for debugging if decisions are sound and no valid formula is found (line 22 and 23).

Algorithm 4.2 describes the necessary modifications of Algorithm 4.1 if one of the descriptions is at gate-level. Parallel simulation is avoided for the reasons described in section 4.5.2. Therefore, a complete path is first simulated in the specification (line 3 in Algorithm 4.2). The information obtained from this path is used to detect equivalences during the subsequent simulation of the implementation (line 9).

Intermediate *dd-checks* are often useful (line 9) if the implementation is at gate-level rather than if both descriptions are at algorithmic- or rt-level. The same entire Boolean expressions assigned to the register bits have to be simulated at gate-level in each symbolic simulation cycle. It is crucial to find relationships of the values of the control registers in the *preceding* cycle in order to detect equivalences in the next cycle between the Boolean expressions at gate-level and the much simpler corresponding terms in the specification at algorithmic- or rt-level. The *final dd-checks* in lines 15 to 21 of Algorithm 4.1 become impractical if the "link" between the specification and the implementation gets lost early on the path: too many intermediate simulation cycles at gate-level have to be considered in the decision diagrams before equivalent terms of the specification and of the implementation are reached, see also the experimental results in section 7.3.

The intermediate *dd-checks* are described with examples in section 6.4. They are used during the path search if no equivalence has been found yet for a term assigned to a *RegVal* at gate-level. It is useful if the user restricts the application of those intermediate tests by simply denoting the control registers. Note that the verification process is automatic and requires no insight from the user.

A practical important property of the symbolic simulator is its good debugging

Algorithm 4.2 Symbolic simulation at gate-level

1. *line 1 to 4 in Algorithm 4.1*
2. **repeat**
3. to_decide:= { - simulate act_case$_{spec}$ until next condition depending on initial *RegVals*; - reduce act_case$_{spec}$ accordingly; - return condition to decide }
4. **if** to_decide is found **then**
5. push (to_decide,act_case$_{spec}$,impl) rem_cases;
6. deny(to_decide);
7. **until** to_decide not found;
8. **repeat**
9. to_decide:= { - simulate act_case$_{impl}$ until next condition depending on initial *RegVals*; - reduce act_case$_{impl}$ accordingly; - perform intermediate *dd-checks* if necessary; - return condition to decide }
10. **if** to_decide is found **then**
11. push (to_decide,NIL,act_case$_{impl}$) rem_cases;
12. deny(to_decide);
13. **until** to_decide not found;
14. *line 11 to 25 in Algorithm 4.1*

support. A complete error trace can be generated for a counterexample since all information about the symbolic simulation of the relevant path is available. For example, it turned out that a report is helpful which summarizes the different microprogram-steps or the sequence of instructions carried through the pipeline registers. Note that only a counterexample in the initial *RegVals* would be available if formulas were canonized. Information from simulation can also be useful if the descriptions are equivalent. Aggregated results about the simulation of all paths are more interesting in this case. For instance, a report about never taken branches of *if-then-else*-clauses turned out to be helpful. It indicates redundancy which may be not detected by logic minimizers.

Verification goals such as property verification can be checked without modifying Algorithm 4.1 and 4.2. They can be reduced to a comparison of *RegVals* as described in section 2.7. Intermediate *RegVals* can easily be checked, too. Only the set of *RegVals* to be compared in line 11, 14, and 15 of Algorithm 4.1 has to be extended in this case.

Chapter 5

Detecting Equivalences of Terms

The equivalence detection on the fly is not complete since it would be too time-consuming to check all possible equivalences of terms. On the other hand, it should be sufficiently powerful so that in most cases the more accurate, but slower *dd-checks* described in chapter 6 are not required. These should only reveal special cases of equivalence which seldom occur or are hard to detect. Note that one reason for the inferior speed of the decision diagram based *dd-checks* is that a backtracking of the simulation is required. All other techniques use in general just the current state of the *EqvClasses* of the direct arguments to detect equivalences between terms; i.e., they avoid a time-consuming backtracking of the expression trees.

Section 5.1 describes the general equivalence detection which can be used for all functions. The rest of the chapter except section 5.10 is structured according to the function type of a term. Equivalence detection for Boolean functions is discussed in section 5.2. Bit-vector functions take bit-vectors as arguments and return a bit-vector or one bit as a result. The most important equivalence detection techniques implemented for bit-vector functions are described in the following sections:

- section 5.3: arithmetic functions, e.g., addition, multiplication, or subtraction; note that some arithmetic functions are transformed during pre-processing, e.g., a left-shift shifting in 1 is transformed into a combination of *bit-selection* and concatenation `lsh(a,1)` $\rightarrow$ `a[30:0]&1`; section 5.3 describes the equivalence detection for addition as a representative of arithmetic functions in detail;
- section 5.4: multiplexers are interpreted as functions with N control bits which select one of 2^N data words;
- section 5.5: comparison functions, e.g., $<$ or $>=$;
- Section 5.6: concatenations of terms which occur often at gate-level since the corresponding register assignments are obtained during pre-processing by concatenating the respective (in general complex) Boolean expressions;

Section 5.7: *bit-selections* (e.g., `ir[7:4]`), which are considered as function invocations;
Section 5.8: `unknown`-terms, see below;
Section 5.9: memory operations, i.e., `store`- and `read`-operations; equivalence detection copes with distinct order of memory operations and was first presented and compared to other approaches in [RHE99].

Only the general techniques presented in section 5.1 are applied on uninterpreted bit-vector functions, e.g., user-defined functions[1]. A special case are `unknown`-terms which are guaranteed to be neither $\cong_{\mathcal{C}}$ nor $\ncong_{\mathcal{C}}$ to another term; this function allows the user to leave implementation dependent parts of the design unspecified or unconsidered.

Equivalence detection for Boolean operations *on bit-vectors* is similar to the corresponding techniques for Boolean operations on bits. However, only a part of the simplification techniques presented in section 5.2 can be applied to bit-vectors.

Finally, section 5.10 describes how the equivalence between a term and a constant caused by a set of inequivalences to other constants and the restricted domain of the term is detected. The type of the functions is summarized in appendix 9.6.

Note that the results of the equivalence detection techniques are marked with few exceptions at the *EqvClasses*. Symbolic terms are never manipulated, e.g., by canonizing or rewriting them, see section 2.1. No unique representation is required which easily allows to add new equivalence detection techniques and which permits a hierarchical equivalence detection according to the principle of Hennessy and Patterson [HP96]: "*Make the common case fast*".

5.1 General Equivalence Detection

5.1.1 Checking Equivalence of Two Terms

Equivalence detection methods developed for a specific function are typically faster and more powerful than general approaches. However, general techniques have to be provided since no function-specific rule may apply or no specific technique exists, e.g., for user-defined functions.

A very general rule is that if the outer function symbol of two terms is the same and all arguments are pairwise equivalent, i.e., the *EqvClasses* of the arguments are pairwise identical then the two terms are equivalent:

$$a_n \cong_{\mathcal{C}} b_n \wedge \cdots \wedge a_0 \cong_{\mathcal{C}} b_0 \Rightarrow f(a_n, \ldots, a_0) \cong_{\mathcal{C}} f(b_n, \ldots, b_0) \qquad (5.1)$$

[1] User-defined functions are replaced during *IDS-to-EDS*-translation if an equivalent expression using known functions is provided.

A weaker condition than Equation 5.1 can be used if the function is symmetric.[2] Basically, it is sufficient to exhibit a permutation of the arguments such that Equation 5.1 applies. *Practically*, testing if any argument has an equivalent counterpart *in one direction* is not sufficient since the number of arguments can vary, e.g., `x+1+1` $\not\cong_{\mathcal{C}}$`x+1` and the same argument can be used twice, e.g., `x+1+1` $\not\cong_{\mathcal{C}}$`x+1+2`.[3] Therefore, the *occurrences* of the *EqvClasses* of the arguments $a_n, \cdots, a_0$ and $b_m, \cdots, b_0$ have to be the same for both terms:

$$
\begin{aligned}
&ArgEC_A := (EqvClass(a_n), \cdots, EqvClass(a_0)) \\
&ArgEC_B := (EqvClass(b_m), \cdots, EqvClass(b_0)) \\
\\
&(f \textit{ is symmetric}) \ \wedge (n = m) \wedge \\
&(\forall x_i \in ArgEC_A : \#occur\,(x_i, ArgEC_A) = \#occur\,(x_i, ArgEC_B)) \\
&\quad \Rightarrow f\,(a_n, \ldots, a_0) \cong_{\mathcal{C}} f\,(b_m, \ldots, b_0)
\end{aligned}
\tag{5.2}
$$

$\#occur(e, \mathcal{S})$ determines how often the element e occurs in the list $\mathcal{S}$. The checks are simplified, if the number of arguments of a function is fixed.

Note that equivalence of the arguments as described by Equation 5.1 and 5.2 need not be a necessary condition for equivalence if other function-specific properties apart from symmetry are considered.

5.1.2 Determining the Set of Candidates

The general equivalence detection techniques require a set of candidates to check equivalence to a new term. Note that

- the specific techniques often also require such a set, and
- the general techniques are mostly used if no function-specific rule applies. Therefore, we use the concatenation as example below, although specific equivalence detection techniques exist for this function.

For user-defined functions or functions, which are not used frequently, all terms with the same function symbol found during simulation on a path are collected. Note that this set of candidates consists in general only of a small fraction of all terms with this function symbol in the whole description since it is *path-dependent*.

However, this approach is inefficient for frequently used functions, especially concatenation and single-*bit-selection*, which occur particularly often at gate-level. A smaller set of candidates is determined for those functions by another approach which examines the *EqvClasses* of the arguments. Consider first a function with a single argument. Two terms are equivalent if the *arguments* are

[2] The techniques described in this section are not described as "uninterpreted" because the symmetric property is employed.

[3] Checking also the opposite direction is less efficient than testing Equation 5.2.

in the same *EqvClass*. Therefore, candidates can be determined by evaluating the *EqvClass* of the *argument* of a new term, i.e., candidates must

i. have an argument which is a member of this *EqvClass*,

ii. use the same function symbol, and

iii. have been found on the current path.

Each of these terms is equivalent to the new term for functions with only one argument. Otherwise the first property must hold for each argument, considering whether the function is symmetric or not.

The set of candidates can be determined easily since the information about which functions use a term as argument is marked at the term during preprocessing. For every $term_i$, the set of terms is determined which use $term_i$ as argument. Different sets are built

- for different function symbols, and
- for asymmetric functions additionally for each position of the argument, e.g., the terms in $\texttt{cat}^{arg1}$ are concatenations which use $term_i$ as first argument. `cat` is the abbreviation for the concatenation, i.e., & in VHDL-notation.

Example 5.1
The set $\texttt{cat}^{arg1}$ *of the term* `ir[4]` *in Fig. 5.1 is* {`ir[4]&y`, `ir[4]&x`}*, the set* $\texttt{cat}^{arg2}$ *of the term* $\texttt{b}_1^i$ *is* $\{\texttt{a}_1^i\texttt{[4]\&b}_1^i\}$*, and the set* $\texttt{cat}^{arg1}$ *of* `x` *is empty. The other sets are determined correspondingly. Assume that both terms in the specification have been found,* `ctrl="000"` *holds, and the term* $\texttt{a}_1^i\texttt{[4]\&b}_1^i$ *in the implementation is checked now.*

$\{\texttt{a}_1^i\texttt{[4], ir[4]}\}$ *are in the EqvClass of the first argument* $\texttt{a}_1^i\texttt{[4]}$*. Unifying the corresponding sets, i.e., the* $\texttt{cat}^{arg1}$*-sets of* $\texttt{a}_1^i\texttt{[4]}$ *and* `ir[4]` *results in* {`ir[4]&y`, `ir[4]&x`, $\texttt{a}_1^i\texttt{[4]\&b}_1^i$}*. In the EqvClass of the second argument are* $\{\texttt{b}_1^i\texttt{, x}\}$ *and the corresponding unified* $\texttt{cat}^{arg2}$*-set is* $\{\texttt{ir[4]\&x}, \texttt{a}_1^i\texttt{[4]\&b}_1^i\}$*. Both arguments have to be equivalent for the equivalence of two concatenations. The intersection of the two sets returns the equivalent terms, which is* {`ir[4]&x`} *in this case.*[4]

The second approach to determine candidates presented above for concatenation can be used also for other functions. However, it is less efficient if the function-specific equivalence detection techniques are mostly successful (e.g., for Boolean- or arithmetic-functions) and/or if only few terms of the same function are encountered *on the same path*. Nevertheless, experimental results have demonstrated that the second approach is useful, if the general equivalence detection techniques are applied to concatenation or *bit-selection*. This approach

[4]Equivalence detection was invoked for $\texttt{a}_1^i\texttt{[4]\&b}_1^i$, i.e., this term is excluded from the intersection.

Specification

```
x1s←ir[4]&y;
y1s←ir[4]&x;
```

Implementation

```
a1i←ir;
if ctrl="000"
    then b1i←x;
    else b1i←y;
...
x1i←a1i[4] & b1i;
...
```

Fig. 5.1: Example for the general equivalence detection technique

would become slow, if the *EqvClass* of an argument has many members which is common for the *EqvClasses* of the constants 0 or 1. But these corner-cases are mostly considered separately by the specialized equivalence detection techniques described in the following sections.

5.2 Boolean Functions

Detecting equivalences of Boolean functions is especially important if one of the descriptions is given at gate-level. The techniques used for Boolean functions also never rewrite or canonize terms. The knowledge about the *EqvClasses* of the direct arguments is used instead of tracing the Boolean expression trees.

As for all other functions, first properties which are fast to identify and which often occur are checked. For Boolean functions, first constant bits are determined. For example, simplification of an `and`-term is obvious if one argument is equivalent to 0 or only one argument is not equivalent to 1. Searching for constants is not always sufficient.

Example 5.2
The relationship $(ak[0]\ nand\ 1)\ and\ (not(ak[0])\ nor\ 0) \cong_C 0$ *has to be detected in Fig. 5.2 to reveal the constant value of* $\texttt{res}_1^i$. *The simplifications of the compilers during pre-processing cannot consider this relationship since it is path-dependent and a result of the previous sequential assignments. Constructing and evaluating*

Specification

```
if x = "0110" then b1s ← ak;
  else ...
if ...  then ...
  else c1s ← ak[3:0];
```

Implementation

previously detected: $\texttt{b}_1^i \cong_C \texttt{b}_1^s$, $\texttt{c}_1^i \cong_C \texttt{c}_1^s$

```
res1i ←(b1i[0] nand x[1]) and
       ((not c1i[0]) nor x[0]);
```

Fig. 5.2: Example for equivalence detection for Boolean functions

the expression $(a\ nand\ 1)\ and\ ((not\ a)\ nor\ 0)$ *is feasible but violates the recursive scheme of equivalence detection: just the information of the EqvClasses of the direct arguments is evaluated in order to avoid a slowdown of the simulation if the depth of the Boolean expressions is greater than in this simple example.*

The difficulty of Example 5.2 is that some of the subterms are not constant. But those subterms are either equivalent to `ak[0]` or to `(not ak[0])`, i.e., this means that they are *positive-* or *negative-bit-equivalent* to `ak[0]`. This information can easily be remarked at the *EqvClasses* during symbolic simulation:

- $\texttt{b}_1^i\texttt{[0]}$ is equivalent to `ak[0]` and `x[1]` $\cong_{\mathcal{C}}$ 1; therefore, the `nand`-term is identified to be *negative-bit-equivalent* to `ak[0]`;
- the term $\texttt{c}_1^i\texttt{[0]}$ is *positive-bit-equivalent* to `ak[0]`; that is why `(not` $\texttt{c}_1^i\texttt{[0])}$ is *negative-bit-equivalent* and the term `((not` $\texttt{c}_1^i\texttt{[0]) nor x[0])}$ *positive-bit-equivalent* to `ak[0]`; note that the *EqvClasses* of the `nand`-term and of `(not` $\texttt{c}_1^i\texttt{[0])}$ are unified; the propagation of *positive-* or *negative-bit-equivalence* has to consider consecutive selections to identify that $\texttt{c}_1^i\texttt{[0]}$ is *positive-bit-equivalent* to `ak[0]` since $\texttt{c}_1^i$ $\cong_{\mathcal{C}}$ `ak[3:0]`;
- the arguments of the `and`-term are *positive-* and *negative-bit-equivalent* to the same bit and, therefore, both can never be satisfied.

Definition 5.1 (*positive-* **or** *negative-bit-equivalence*)
Let `Bit` *be a single-bit term or the* `i`*-th bit of a term, i.e.,* `term[i]`*. The bit-selection* `term[i]` *need not appear in the descriptions.*[5] *A single-bit term is positive-bit-equivalent (negative-bit-equivalent) to* `Bit` *if they are* $\cong_{\mathcal{C}}$ ($\ncong_{\mathcal{C}}$).

The expressions *positive-* or *negative-bit-equivalent* are used in this work although the underlying relationship could be expressed using $\cong_{\mathcal{C}}$ and $\ncong_{\mathcal{C}}$. The reason is that this information has to be marked mostly separately at the *EqvClasses* since the corresponding *bit-selections* of terms do not appear explicitly in the descriptions, e.g., `not(ak[0])`. Therefore, unifying *EqvClasses* or marking inequivalence is not possible. Creating artificially terms for all possible *bit-selections* during pre-processing and building *EqvClasses* for them would be too costly or inefficient.[6]

Remarking *positive-* or *negative-bit-equivalence* at the *EqvClasses* is not only used to detect contradictions. For example, if an argument of an `and`-term is *negative-bit-equivalent* to a bit and all other arguments are equivalent to 1, then the `and`-term is equivalent to the negation of this bit. More applications are straightforward, see below.

In the following, `and`-terms are taken as example for Boolean functions. Let $(and\ x_1, \cdots, x_n)$ be an `and`-term in prefix notation with n arguments. The applied rules (with descending priority) are described in Fig. 5.3.

The implementation of the equivalence detection is sketched in Algorithm 5.1. Note that experimental evidence was used to optimize the application of the

[5] *Positive-bit-equivalence* has to be marked in these cases. Otherwise the information of the *EqvClass* is sufficient.

[6] Another implementation advantage is that it is not necessary to trace the member list of an *EqvClass* to find the *positive-* or *negative-bit-equivalent* term.

i. $\exists x_i : x_i \cong_{\mathcal{C}} 0 \quad \Rightarrow (and\ x_1, \cdots, x_n) \cong_{\mathcal{C}} 0$

ii. $\forall x_i : x_i \cong_{\mathcal{C}} 1 \quad \Rightarrow (and\ x_1, \cdots, x_n) \cong_{\mathcal{C}} 1$

iii. $\forall_{i \neq j}\ x_i : x_i \cong_{\mathcal{C}} 1 \quad \Rightarrow (and\ x_1, \cdots, x_n) \cong_{\mathcal{C}} x_j$

iv. $(\exists x_i : x_i \cong_{\mathcal{C}} a) \wedge (\exists x_j : x_j \ncong_{\mathcal{C}} a) \ \Rightarrow (and\ x_1, \cdots, x_n) \cong_{\mathcal{C}} 0$

v. apply rule described by Formula 5.2 on page 67

Fig. 5.3: Rules applied to find equivalent and-terms

Algorithm 5.1 Detecting Equivalences of AND-terms

```
input term
 1. let const-of-args := {}, arguments := arguments-of(term)
 2. foreach arg ∈ arguments do
 3.       if constant(arg) = 0 then
 4.            propagate-bit-equivalence(arg,term);
 5.            eqvclass-merging(term,0);
 6.            return;
 7.       elsif NOT(constant(arg) = 1)
 8.             push(arg,args-not-const); od
 9. if args-not-const = {} then
10.       propagate-bit-equivalence(arguments,term);
11.       eqvclass-merging(term,1);
12. elsif |args-not-const|=1
13.       propagate-bit-equivalence(first(args-not-const),term);
14.       eqvclass-merging(term,first(args-not-const));
15. elsif positiv-and-negativ-bit-equivalent(arguments)
16.       propagate-bit-equivalence(select-best(args-not-const),term);
17.       eqvclass-merging(term,0);
18. else check-sym-fn-without-const(args-not-const,term,AND);
```

rules with regard to simulation speed. Furthermore, the use and the propagation of the information about *positive-* or *negative-bit-equivalence* is described. Some programming optimizations are omitted for clarity. All arguments which are not $\cong_{\mathcal{C}}$ to 0 or 1 are collected in line 2 to 8. The `and`-term is $\cong_{\mathcal{C}}$

- to 0 if one argument is $\cong_{\mathcal{C}}$ to 0 (lines 4 to 6);
- to 1 if all arguments are $\cong_{\mathcal{C}}$ to 1 (line 10 and 11);

- to one argument if all other arguments are $\cong_{\mathcal{C}}$ to 1 (line 13 and 14);
- to 0 if there are two arguments, which are *positive-* respectively *negative-bit-equivalent* to equivalent bits (line 16 and 17); this is checked by comparing two sets which contain the *positive-bit-equivalent-* respectively *negative-bit-equivalent*-bits of the arguments. If there is a pair in the same *EqvClass* then the two corresponding arguments cannot be simultaneously one, i.e., the `and`-term is equivalent to 0.

Otherwise, the general equivalence detection for symmetric functions is called in line 18 only with the non-constant arguments (all other arguments are equivalent to 1).

Positive- or *negative-bit-equivalence* has to be propagated even if the term is equivalent to a constant (line 4 and 10) in order to detect equivalences of concatenations (an example is given below).[7] A heuristic is used if the arguments are *positive-* or *negative-bit-equivalent* to different bits. If all but one argument are constant or eliminate each other because they are *positive-* and *negative-bit-equivalent* to the same bit then the information of the remaining argument is propagated. Otherwise the *positive-* or *negative-bit-equivalence* to be propagated is selected according to the following priorities:

i. the bit is equivalent to a bit of an initial *RegVal*;

ii. the bit is not in an *EqvClass* with a constant;

iii. if two bits are *bit-selections* from two terms, then the one is preferred where not all bits of the selected term are constant.

Criterion ii. and iii. consider that the register assignments at gate-level are concatenations of complex Boolean expressions. Correct propagation is crucial to identify equivalence to simpler terms of the specification at top level.

Example 5.3
The equivalence of $\texttt{res}_1^s$ *and* $\texttt{res}_1^i$ *has to be detected in Fig. 5.4. The hidden*

Specification	**Implementation**
if (ir[0]='1' and mad="0101")	$\texttt{res}_1^i \leftarrow$((...) or (not (mad[3]) and ir[3])) &
then $\texttt{res}_1^s \leftarrow$ir;	((...) or (mad[2] and ir[2])) &
else ...	((...) or (not (mad[1]) and ir[1])) &
	((...) or (mad[0] and ir[0])) &

Fig. 5.4: Priority example for propagating *positive-* or *negative-bit-equivalence*

terms in brackets (...) *on the implementation side summarize the assignments on other paths of the specification and are assumed to be equivalent to 0 on the*

[7] Note that also the arguments which are equivalent to constants are considered in line 15.

current path. Positive- or negative-bit-equivalence to `ir[3]`, `ir[2]`, *and* `ir[1]` *are propagated for the most significant bits of* $\texttt{res}_1^i$ *since the other arguments of the* `and`*-terms* (`mad[3]`, `mad[2]`, `mad[1]`) *are equivalent to constants. But* both *arguments* `mad[0]` *and* `ir[0]` *of the least significant bit are equivalent to 1. However, it makes more sense to propagate* `ir[0]` *following criterion iii: all bits of term* `mad` *are constant; i.e., equivalence to* `mad` *and the equivalent constant* `0101` *could be detected after concatenation without the knowledge of positive- or negative-bit-equivalence. Therefore,* `ir[0]` *is propagated and equivalence to* `ir` *is detected after concatenation.*

The algorithms of equivalence detection for the Boolean functions `or`, `nand`, `nor`, `xor` and `not` are derived accordingly to Algorithm 5.1. Note that, for example, the union of the *EqvClasses* in line 14 is not feasible for a `nand`-term. Standard cells or other Boolean functions are currently broken during pre-processing using those basic Boolean functions. For example, the `AO2`-standard-cell of the Alcatel™ MTC45000-library is transformed into `(A and B) nor (C and D)`. Simulation speed can be optimized by providing specialized equivalence detection routines for those standard cells, too.

5.3 Arithmetic functions

Many arithmetic functions used in *hardware*-designs are modulo operations, either explicitly or implicitly. Equivalence detection for the addition with carry-input but *without* carry-output `adcmod(a,b,carry)` is presented as an example in the following. Algorithm 5.2 gives an overview.

If all the arguments of an `adcmod`-term are constant (line 1) then the constant result of the term is calculated and the corresponding *EqvClasses* are unified (line 2 and 3). Note that this may make the dynamic creation of an *EqvClass* necessary (see section 4.3). *EqvClasses* are built during pre-processing only for constants appearing explicitly in the descriptions, but the result of line 2 can be a new constant.

If the `carry` of the term is equivalent to 0, i.e., it is irrelevant then the equivalence detection for symmetric functions for the addition *without* input carry (`addmod`) is called (line 11). Moreover, if one of the summands is equivalent to 1 then the result is the same as incrementing the remaining non-constant argument, i.e., any `incmod`-term with an $\cong_C$ argument is equivalent (line 5 and 7). The same holds if the carry is equivalent to 1 and one of the summands is equivalent to 0 (line 12 and 14). Note that although the equivalence detection is reduced in line 5, 7, 11, 12, and 14 to check equivalence for `incmod` respectively `addmod`, equivalence to another `adcmod`-term will be still detected since its arguments would satisfy the same properties.

If the carry and one of the summands is equivalent to 0 then the *EqvClasses* of

Algorithm 5.2 Detecting Equivalences for Addition without Carry-output

input `term ADCMOD(A,B,carry-in)`

1. **if** $\forall_{i \in args}$ `:const-of(`i`)` **then**
2. `const_result :=` $\left(\sum_{i \in args} \texttt{const-of}(i)\right)$ mod $\left(2^{length-of(\texttt{term})}\right)$`;`
3. `eqvclass-merging(term,const_result);`
4. **elsif** `const-of(carry-in)=0` **then**
5. **if** `const-of(A)=1 AND check-eqv-inc(term,B)` **then**
6. `eqvclass-merging(term,equivalent-inc-term(term,B));`
7. **elsif** `const-of(B)=1 AND check-eqv-inc(term,A)` **then**
8. `eqvclass-merging(term,equivalent-inc-term(term,A));`
9. **elsif** `const-of(A)=0` **then** `eqvclass-merging(term,B);`
10. **elsif** `const-of(B)=0` **then** `eqvclass-merging(term,A);`
11. **else** `check-equivalence(term,ADDMOD,A,B);`
12. **elsif** `const-of(carry-in)=1 AND const-of(A)=0 AND check-eqv-inc(term,B)`
13. **then** `eqvclass-merging(term,equivalent-inc-term(term,B));`
14. **elsif** `const-of(carry-in)=1 AND const-of(B)=0 AND check-eqv-inc(term,A)`
15. **then** `eqvclass-merging(term,equivalent-inc-term(term,A));`
16. **else** `check-asym-fn(A,B,carry-in,ADCMOD);`

the `adcmod`-term and the remaining non-constant argument are unified (line 9 and 10). Otherwise the general equivalence detection technique described in section 5.1 is used *considering the carry* (line 16). Note that the *specific* equivalence detection for `addmod` is called in line 11 and *not* the general equivalence detection techniques as in line 16.

Equivalence of successive additions is considered by accumulating the constants and collecting the non-constant arguments. For example, if $\texttt{x}_1^i$ `←a+b+4` holds then `5+`$\texttt{x}_1^i$ has the accumulated constant `9` and the *positive* non-constant part `{a,b}`.[8] Two terms are equivalent if the non-constant parts are equivalent and the accumulated constants are equal, which has to be considered in line 16 and 11 as well as in `check-eqv-to-inc`. An extension of this concept to include subtraction etc. must carefully consider overflows and underflows, which limits the application substantially.

Unification of the *EqvClasses* in line 5 to 11 can lead to an *EqvClass* with terms of different lengths if a carry-output is considered. For example, if equivalence of an `add`-term *with* carry-output is tested, then the unification in line 9 causes that

[8] Negative non-constant parts result from subtractions.

the argument `B` with bit-vector length n is in the same *EqvClass* as the `add`-term with length *n+1*. Different bit-vector lengths in one *EqvClass* are accepted iff all leading bits of the terms with greater length are guaranteed to be $\cong_{\mathcal{C}}$ to 0 or the *EqvClass* contains a constant. This implicit notation is also considered during the *dd-checks* described in chapter 6.

5.4 Multiplexer

Multiplexers are interpreted as functions with N control bits which select one of 2^N data words. A transformation into an adequate *if-then-else*-clause is feasible, but blows up the descriptions: the size of the structure doubles with each additional control bit. This can lead to term-size explosion in other approaches, if the overall formula is built in advance and verified afterwards, e.g., if a big ROM is used, see section 3.3. An alternative is to interpret multiplexers as functions:

$$mpx_N(\mathtt{C},\mathtt{D}) = \mathtt{d}_{(2^{N-1}\cdot \mathtt{c}_{N-1}+2^{N-2}\cdot \mathtt{c}_{N-2}+\ldots+\mathtt{c}_0)} \tag{5.3}$$

$\mathtt{C}$ and $\mathtt{D}$ are bit-vectors with the bits $\mathtt{c}_0$ to $\mathtt{c}_{N-1}$ and $\mathtt{d}_0$ to $\mathtt{d}_{2^N-1}$

Equation 5.3 subsumes that each control bit is equivalent to either 1 or 0. The *EqvClass* of the `mpx`-term and of the selected data word on the right-hand side can be unified in this case. It is not possible to decide which data word is selected if one of the control bits is not in the *EqvClass* of 1 or 0. An application of the general equivalence detection techniques (section 5.1) is not efficient in this case. The term has $2^N + N$ arguments and equivalence detection is rarely successful since *all* data words and control bits of two `mpx`-terms have to be equivalent.

Therefore, a decision about the value of the control bits is forced for each `mpx`-term by introducing a *single* special *if-then-else*-clause in front of each `mpx`-term during pre-processing. Fig. 5.5 (b) and (c) show the internal representation of a 8:1 multiplexer before and after transformation during pre-processing. The equivalent structural description is given in Fig. 5.5 (a). Note that the data words `d0` to `d7` can be bit-vectors.

This special *if-then-else*-clause guarantees that a single data word is selected. The (Boolean) arguments of the predicate `mpx-or` are transformed into *CondBits* during pre-processing. The `mpx-or` is interpreted during simulation as a disjunction (`or`), i.e., the `else`-branch is only reached if all arguments are equivalent to 0. The only difference is that if one of the arguments is identified to be equivalent to 1 (*CondBit* is *true*) then `or` evaluates no more arguments and performs, therefore, no additional case-splits, see section 4.4. In contrast, `mpx-or` forces case-splits until *all* arguments $\mathtt{c}_0,\ldots,\mathtt{c}_{N-1}$ are equivalent to either 1 or 0. Therefore, it is guaranteed that all control bits are equivalent to constants when reaching the `mpx`-term, i.e., a particular data word is selected. `mpx-or` results in *false* iff all control signals are equivalent to 0. The data word `d0` of the multiplexer is selected in this case, see the `else`-branch in Fig. 5.5 (c).

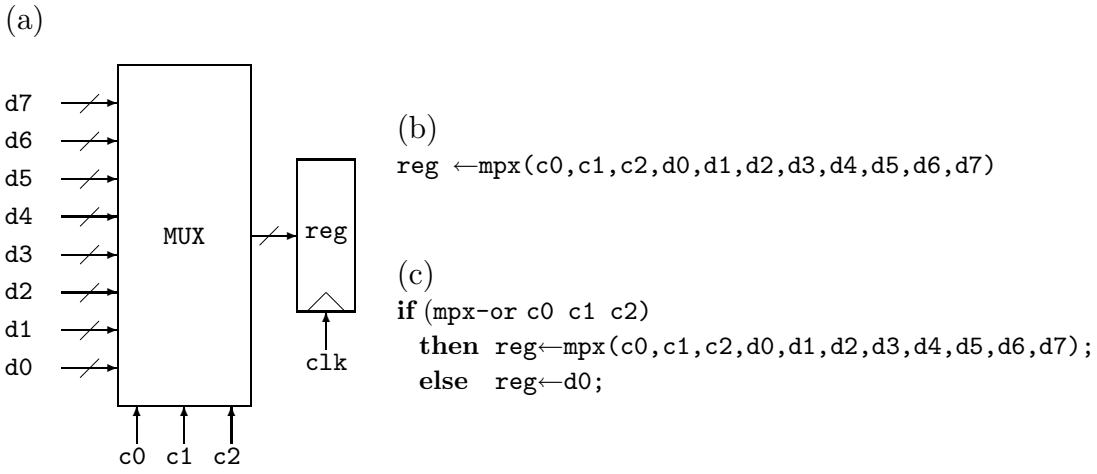


Fig. 5.5: Transformation of multiplexers

5.5 Comparison

Comparisons, i.e., >, <, >=, and <=, are mostly used in conditions. Some comparisons are transformed during pre-processing and are, therefore, not discussed in the following:

- $a \not\equiv b$ is transformed to `not(`$a \equiv b$`)`;
- comparisons on bits can be reduced to Boolean formulas, e.g., $\texttt{bit}_a \leq \texttt{bit}_b$ is the same as `not(bit`$_a$`) or bit`$_b$;
- equivalences ($\equiv$) in conditions are transformed into *CondBits*, see section 4.4; equivalence detection outside of conditions is straightforward using the information of the *EqvClasses*.

A comparison can be decided if both arguments are equivalent to constants by simply comparing the corresponding constants. If the arguments are $\cong_{\mathcal{C}}$ then $\leq$ and $\geq$ are *true* while $<$ and $>$ are *false*. Note that no decision can be derived if the arguments are $\ncong_{\mathcal{C}}$.

Otherwise information about the range of the arguments, marked as *valuebounds* at the *EqvClasses*, is evaluated. The range of terms is mainly restricted by deciding conditions, e.g., `a<30`. But arithmetic operations or concatenations provide also information about the range of terms, e.g., the four-bit vector $00\&a_1\&a_0$ is guaranteed to be less than 4. The relationship is notified at the *EqvClasses* of the arguments which contain no constant[9] as a *valuebound*, e.g. (< 30), $(\geq c)$, or $(< b + e)$. An *EqvClass* can have multiple *valuebounds*. Noting those *valuebounds* is only required for the comparisons >, <, $\leq$, and $\geq$ since the information about $\cong_{\mathcal{C}}$ or $\ncong_{\mathcal{C}}$ can be obtained directly from the *EqvClasses*.

[9] *Valuebounds* for *EqvClasses* containing constants are redundant since the value is fixed.

Notifying the *valuebounds* at the *EqvClasses* permits to find quickly all the decisions about previous comparisons that might be relevant for a new comparison. The *valuebounds* describe all previous comparisons where one of the arguments is $\cong_{\mathcal{C}}$ to an argument of the new comparison. Two terms are compared by examining pairwise the *valuebounds* of the corresponding *EqvClasses*, which can be incompatible, compatible or indifferent concerning the relevant comparison operator.

Example 5.4
Consider the comparison $x < y$ with the valuebounds $\mathcal{V}_x$ and $\mathcal{V}_y$ of the corresponding EqvClasses. If $\mathcal{V}_x = \{(< c), (> d)\}$ holds then

- *the comparison is true for $\mathcal{V}_y = \{(> c)\}$,*
- *the comparison is false for $\mathcal{V}_y = \{(< d)\}$, and*
- *no decision is possible for $\mathcal{V}_y = \{(\leq c), (\geq d)\}$.*

Note that the *EqvClasses* of the arguments are used when comparing *valuebounds*, e.g., the *valuebounds* $(< d)$ and $(> e)$ are detected to be mutual exclusive, if d and e are in the same *EqvClass*.

The comparison is simpler if one of the arguments is equivalent to a constant. Otherwise all combinations of *valuebounds* of the left-hand and right-hand side of a new comparison have to be considered. Comparing only the argument directly with the *valuebounds* of the opposite side is insufficient. For example, assume that the term x is not in the *EqvClass* of c or d in Example 5.4. Comparing x directly to $\mathcal{V}_y$ does not reveal that $x < y$ is *true/false* in the first two cases. Equivalence detection may be used recursively, e.g., the comparison $a_2^s < b_2^s$ assuming the *valuebounds* $a_2^s < x_1^s$ and $b_2^s > y_1^s$ is satisfied if $x_1^s \leq y_1^s$ holds.

Valuebounds are not only generated by deciding conditions but also in the following cases:

- bits are selected from a term; all *valuebounds* of the term with a *constant* are used to determine the *valuebounds* of the *bit-selection*;

  Example 5.5
  Let `reg` *be a 6 bit register. The least significant bit of* `reg` *has the index 0. The corresponding EqvClass has the valuebound $\{(\leq 8)\}$. The EqvClass of the bit-selection* `reg[5:2]` *gets the valuebound $\{(\leq 2)\}$.*

- if the term `x` with the most-significant bits of a concatenation `x&y` is equivalent to 0, then the concatenation cannot have a value greater than the domain of the term `y` with the least-significant bits;

- if only one argument of an addition is not equivalent to a constant, then the new *valuebounds* can be calculated, if either the addition is not modulo or if no overflow can occur.

The information about constant and non-constant parts of two arithmetic operations to be compared (see section 5.3) is taken into account, but often permit no decision if the operations are modulo. Comparing the accumulated constants is not sufficient even if the non-constant parts are equivalent, e.g., $(a+4 < a+3)$ may hold due to an overflow.

Just as most of the other techniques presented in this chapter, equivalence detection for comparison is not complete. For example, the concept of *value-bounds* may be extended to consider not only conjunctions of range restrictions, but also disjunctions. There exist more possibilities for the generation of *value-bounds* which can be integrated into the symbolic simulator. Again, the trade-off between increasing accuracy and simulation speed has to be considered.

5.6 Concatenation

Detecting equivalences of concatenations is particularly crucial if descriptions at algorithmic- or rt-level are compared to gate-level descriptions. The assignments to registers at gate-level are obtained during pre-processing by concatenating the respective (in general complex) Boolean expressions bit_i, i.e., `reg`$\leftarrow (bit_n\&(\ldots\&(bit_2\&(bit_1\&bit_0))\ldots)$, see also appendix 9.4. The parentheses consider the recursion scheme since the concatenation takes only two arguments. For example, first $bit_1\&bit_0\cong_{\mathcal{C}}$`pc[1:0]` is detected during simulation if the expression assigned to `reg` is equivalent to `pc`, then $bit_2\&(bit_1\&bit_0)\cong_{\mathcal{C}}$`pc[2:0]` and so on, see also the example below. Note that the *bit-selections* may not appear explicitly as terms in the descriptions.

Section 5.2 described how knowledge about *positive-* or *negative-bit-equivalence* is propagated for Boolean terms. This information is used to detect equivalences after concatenating the bits.

Example 5.6
Fig. 5.6 gives a realistic example. The concatenation is expressed in IDS-format recursively, i.e., `(cat X (cat Y Z))` *means* `X & Y & Z` *in VHDL-notation. The internal representation in prefix-form is used only in this example to demonstrate equivalence detection for concatenation, i.e., the VHDL-operator '&' is used in the other sections for better readability. The structural description in Fig. 5.6 (b) illustrates the implementation.*[10] *The* `reset` *input (generated by the synthesis tool) does not exist in the specification and is assumed to be set to* `0`*. The least significant bit of* `ctrl` *is on the right-hand side, i.e., if* `ctrl="01"` *holds then* `ctrl[0]` *is set to* `1`*.*

`OUT_of_INC` *is a block which computes the increment of the input.* `n537`*,* `n517`*, and* `n516` *are only simulation-cutpoints (not to be confused with the dd-cutpoints*

[10] Standard-cells are broken in Fig. 5.6 (b), e.g., two `or`-gates and the following `nand`-gate represent one cell in the original synthesis result.

(a)

Specification

```
if ( ctrl="01")
    then pc_1^s ←pc;
    else pc_1^s ←pc+1;
```

Implementation

```
n537_1^i := (not ctrl[1]) nand ctrl[0];          ;;only simulation-cutpoint
n517_1^i := (not reset) nand n537_1^i;           ;;only simulation-cutpoint
n516_1^i := (not reset) nand n517_1^i;           ;;only simulation-cutpoint
pc_1^i ←
  (cat((not OUT_OF_INC[7]) or n517_1^i) nand (n516_1^i or (not pc[7]))
      (cat((not OUT_OF_INC[6]) or n517_1^i) nand (n516_1^i or (not pc[6]))
                ...
          (cat((not OUT_OF_INC[1]) or n517_1^i) nand (n516_1^i or (not pc[1]))
              ((not OUT_OF_INC[0]) or n517_1^i) nand (n516_1^i or (not pc[0]))))))))));
```

(b) Structural description of the implementation

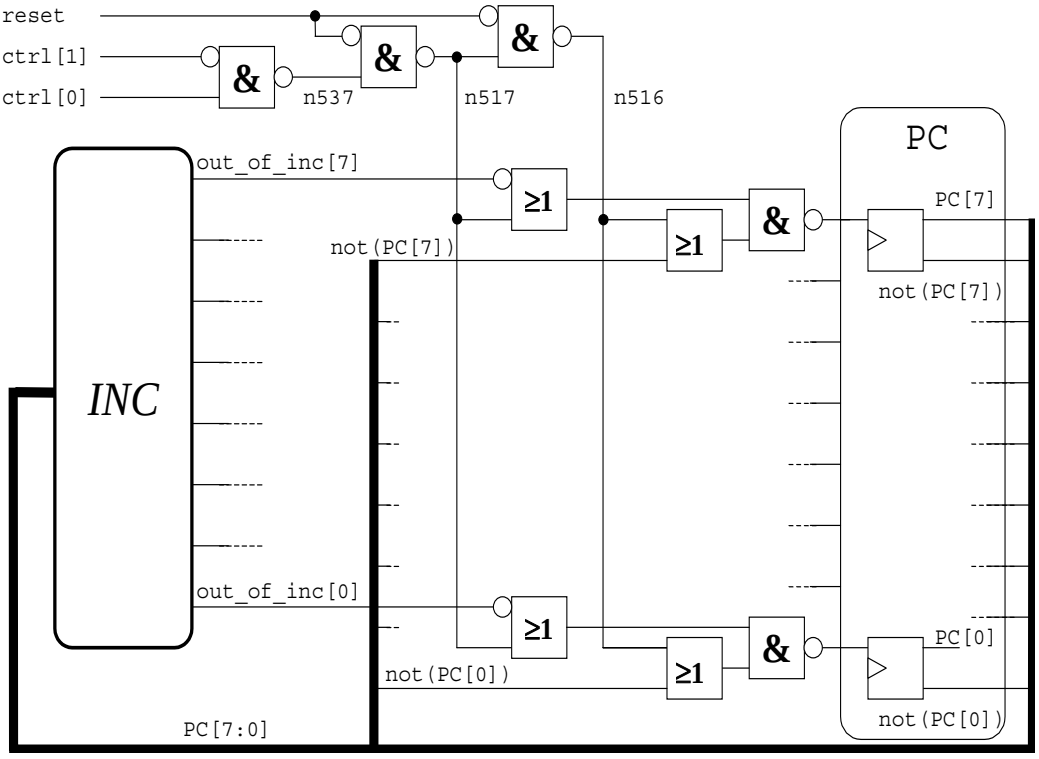


Fig. 5.6: Detecting equivalences after concatenation

of section 6.2). They represent the output of gates in the corresponding gate-level representation with a fan-out greater than one.[11] *Introducing simulation-cutpoints for these signals avoids multiple evaluation of the corresponding expressions, see appendix 9.3 for more details.*

Consider the `then`*-branch in the specification, where* `ctrl` *is equivalent to* `"01"`*. Therefore* $\texttt{n537}_1^i \cong_c \texttt{0}$, $\texttt{n517}_1^i \cong_c \texttt{1}$, *and* $\texttt{n516}_1^i \cong_c \texttt{0}$ *hold. The terms* `((not OUT_OF_INC[`k`]) or n517`$_1^i$`)` *are equivalent to 1 so that the negative-bit-equivalence*

[11] `n537` is only used once in Fig. 5.6, but might be used elsewhere in the circuit.

to `pc[`k`]` *of the second argument* `(n516 or (not pc[`k`]))` *of the* `nand`*-terms is propagated, see section 5.2. Therefore, the result of these* `nand`*-terms is positive-bit-equivalent to* `pc[`k`]`*. The inner* `cat`*-term is equivalent to* `pc[1:0]`*, the second to* `pc[2:0]` *and so on. Finally, the top-level* `cat`*-term and* $\texttt{pc}_1^i$ *are equivalent to* `pc`*. The same procedure is used for the* `else`*-branch of the specification to detect the equivalence of the* `cat`*-term and* `OUT_of_INC` *(i.e.,* `pc+1`*) as well as in Example 5.3 of section 5.2.*

Equivalence detection for concatenation is summarized in Algorithm 5.3. If both arguments are equivalent to a constant (line 2) then simply the constant result is calculated (line 3). Otherwise the constant regions of the term are marked, e.g., which bits of the concatenated expression are equivalent to a constant (line 5). A constant region is described by the lowest respectively the highest bit of the region and the equivalent constant $[upper : lower] = const$. The regions are determined using the corresponding information of the arguments if they are also `cat`-terms. Furthermore, if one of the arguments is equivalent to a constant then the corresponding constant region of the `cat`-term is notified.

Example 5.7
Two terms `b` *(5 bits) and* `a` *(3 bits) are concatenated* `(cat b a)`*. The least significant bits of the concatenation represent* `a`*;*

- *if* `a` *is equivalent to 1 then the* `cat`*-term gets the constant region [2:0]=1;*
- *if* `b` *is also a* `cat`*-term with the constant region [3:1]=0 then the* `cat`*-term gets the constant region [6:4]=0.*

A `cat`*-term can have multiple constant regions. Overlapping regions are unified.*

Marking those regions has two advantages. First, re-checking whether the entire `cat`-term is $\cong_C$ to a constant is faster, e.g., if later on the path some bits are set constant due to a decided condition. Second, deciding conditions consistently is better supported. For example, if the bits 5 to 1 of a `cat`-term `x` are equivalent to 0 then a condition testing `x[3:2]`$\equiv$`1` is false. The information about constant regions is marked at the *EqvClasses*. If two *EqvClasses* are unified, then the compatibility of the constant regions is tested and the new constant regions resulting from both *EqvClasses* are determined. Note that information about equivalence of *single* bits to constants is also provided by the techniques described in section 5.10.

The value of the second argument of a `cat`-term, which represents the least significant bits, and the `cat`-term itself is in any case identical, if the most significant bits are equivalent to 0 (line 6 and 7 in Algorithm 5.3). The unification of the corresponding *EqvClasses* in line 7 leads to an *EqvClass* with terms of different lengths. These differences are generally accepted in our symbolic simulation approach if all leading bits of the terms with greater length are guaranteed to be equivalent to 0, see section 5.3.

Algorithm 5.3 Detecting Equivalences for Concatenation

```
input (cat upper-bits lower-bits)
 1. let lower-const := const-of(lower-bits);
        upper-const := const-of(upper-bits);
 2. if upper-const ∧ lower-const
 3.       eqvclass-merging(term,upper-const·2^length-of(lower-bits)+lower-const);
 4.       check-for-complete-cat(term);
 5. else mark-const-regions(term);
 6.       if upper-const = 0 then
 7.             eqvclass-merging(term,lower-bits);
 8.             check-for-complete-cat(term);
 9.       elsif check-for-complete-cat(term) ;;returns 'true' if equivalent simpler
                                             ;;term found
10.       else check-two-arg-asym-fn(upper-bits,lower-bits,CAT);
```

In line 4, 8, and 9 of Algorithm 5.3 it is tested whether the `cat`-term represents the concatenation of another, simpler term, or at least the *bit-selection* of such a term. This test detects the equivalence of the inner `cat`-terms in Fig. 5.6 to the respective *bit-selections*, e.g., `pc[2:0]`. Furthermore, the equivalence of the top-level `cat`-term to either `pc` or `OUT_of_INC` is detected.

Example 5.8

- *if* $y \cong_{\mathcal{C}}$ `pc[5]` *and* `(cat` $x \cdots$`)` $\cong_{\mathcal{C}}$ `pc[4:0]` *then the concatenation* `(cat` y `(cat` $x \cdots$`))` *is equivalent to* `pc[5:0]`*;*

- *if* $u \cong_{\mathcal{C}}$ `pc[7]` *and* `(cat` $v \cdots$`)` $\cong_{\mathcal{C}}$ `pc[6:0]` *then the concatenation* `(cat` u `(cat` $v \cdots$`))` *is equivalent to the entire register* `pc` *with 8 bits.*

Note that the *bit-selections* need not appear as terms in the descriptions, e.g., there exists not necessarily an *EqvClass* for `pc[5:0]`. Therefore, the information about equivalence to the *bit-selections* has to be notified, i.e., propagated separately. It is important to propagate this information even if a `cat`-term is $\cong_{\mathcal{C}}$ to a constant (line 4 in Algorithm 5.3). Otherwise equivalence to the entire simpler term cannot be detected at the top-level concatenation, see Example 5.3 in section 5.2.

It is *not* efficient to collect the information about equivalences to the bits of `pc` only when the top-level `cat`-term is reached instead of propagating the information successively. One of the principles of symbolic simulation is to avoid tracing the expression trees of the arguments to permit a fast simulation. Therefore, only the information of the direct arguments has to be used.

Finally, the general equivalence detection technique for asymmetric functions is applied in line 10 if all other tests fail.

5.7 Bit-selection

Bit-selections are considered as function invocations. For example, the *bit-selection* `ir[8:3]`, described as `ir(8 downto 3)` in VHDL-notation, is a term distinct to `ir`.[12] The indexes are integers since all indirect selections are considered as memory operations, see section 4.1.5.

The result is constant if the term of the selection, e.g., `ir` is equivalent to a constant or the selected part is overlapped by a constant region. These regions are frequently the result of a concatenation of terms, where one term is equivalent to a constant. Testing whether the *bit-selection* is overlapped is fast since constant regions are explicitly marked for concatenations, see section 5.6. Additionally, the information about the equivalence of single bits to the constants 0 or 1 detected by the techniques described in section 5.10 is used.

If the *bit-selection* is not overlapped entirely by a constant region then possibly partial constant regions of the *bit-selection* are determined. The limits have to be corrected by the lower index of the *bit-selection*. Furthermore, the new constant value has to be calculated if a region is "cut" by the frontiers of the *bit-selection*.[13] Finally, the general equivalence detection techniques described in section 5.1 are applied with some modifications:

- if the *bit-selection* results in a bit-vector (e.g., `ir[8:3]`), then the *constant* indexes are directly compared instead of considering the corresponding *EqvClasses*;

- single-*bit-selection*, e.g., `ir[4]` is considered as a function with only one argument (`ir`). The general equivalence detection uses the second approach described in section 5.1.2 to determine candidates for equivalence checking. The index of the selected bit is considered in the function symbol, e.g., `(bit-selection-4 ir)` instead of `(bit-selection ir 4)` in pre-fix notation. Therefore, it is sufficient to mark the single-*bit-selections* separately for each index during pre-processing at the single argument, i.e., `ir`. An equivalent *bit-selection* is found, if a term in the *EqvClass* of the argument exists, which is used as an argument in a *bit-selection* with the same index. For example, assume that `ir[4]` is examined and `ax` is in the *EqvClass* of `ir`. If a term exists marked as `bit-selection-4` at `ax`, then the *EqvClasses* of this term `ax[4]` and of `ir[4]` are unified, if `ax[4]` has

[12] The internal function symbols `selel` or `selslice1` are used for the selection of one bit or a bit-vector. The abbreviation "*bit-selection*" is used for both in the following.

[13] For example, if bits 3 to 1 of a term `x` are equivalent to 7 then bits 1 to 0 of `x[10:2]` are equivalent to 3.

been found previously on the path. Note that the equivalence detection described above is fast since it is only checked whether one of the members of an *EqvClass* (without constant) has a corresponding marking.

5.8 Unspecified Parts: "unknown"-Terms

The symbolic simulator has to cope with arbitrary functions defined by the user. If no specific detection scheme is provided for a function then at least the general equivalence detection technique for asymmetric functions presented in section 5.1 is applied. Terms which are guaranteed to be neither $\cong_{\mathcal{C}}$ nor $\ncong_{\mathcal{C}}$ to another term are used in two cases where equivalence detection *has to fail*:

- the user does not specify parts of the design. For example, the assignment to a register can be implementation-dependent in some cases, but should not affect the correct behavior of the entire design;
- missing parts in one of the descriptions have to be considered. For example, some bits of registers only exist in the specification but not in the implementation due to optimizations during synthesis. An unknown value has to be assumed for the missing bits to permit a complete concatenation of the register as described in section 5.6.

Example 5.9
The bits `ir[1]` *and* `ir[0]` *are not used in the specification of Fig. 5.7. Therefore, they do not exist in the implementation after synthesis since they are identified by the synthesis tool to be redundant.* `Unknown`*-terms represent them in the implementation to allow a comparison with the* `ir`*-register in the specification.*

Specification

```
ir←a+b;
if ir[5:2]=011 then ...
   elsif ir[5:2]=110
   ...
```

Implementation

```
ir←alu_out[5] & alu_out[4] &
    alu_out[3] & alu_out[2] &
    unknown(37) & unknown(38);
```

Fig. 5.7: Introducing `unknown`-terms for missing bits

Distinct terms can be generated using the special function `unknown` (see Fig. 5.7) for which none of the equivalence detection techniques is applied. Distinct constants are used as arguments to distinguish the different `unknown`-terms.[14] Note that the same effect is achieved by a user-defined function for which only the

[14]This is necessary since each term is replaced during pre-processing for technical reasons by an arbitrary chosen distinct variable (see appendix 9.2) and different *EqvClasses* have to be built for the `unknown`-terms.

general equivalence detection techniques apply. The corresponding terms cannot be equivalent, too, if the arguments are distinct constants. The advantage of `unknown`-terms is that the general techniques are not unnecessarily applied.

Although `unknown`-terms are neither $\cong_{\mathcal{C}}$ nor $\ncong_{\mathcal{C}}$, the same need not hold for terms using `unknown`-(sub)terms as arguments.

Example 5.10
The term `(unknown(9) nand ctrl)` *is equivalent to 1 for* `ctrl`$\cong_{\mathcal{C}}$`0`. *Otherwise the* `unknown`*-function has an impact and no equivalence is detected.*

`Unknown`-terms permit to reveal erroneous assumptions of the designer about irrelevant terms, e.g., if some assignment is replaced by an `unknown`-term. The final *RegVals* of the specification and of the implementation cannot be in the same *EqvClass* if the `unknown`-term has an impact in any way, and the counterexample is reported.

5.9 Memory Operations

5.9.1 Overview

Formal verification often has to cope with memories that have a large size and are addressed indirectly. Symbolic address relationships of the memory operations have to be considered. Addresses are compared in our approach using only the information of the *EqvClasses*. This allows a fast equivalence detection which can cope with complex reorderings of memory operations. Equivalence detection for memory operations was first presented in [RHE99].

Example 5.11
The two descriptions in Fig. 5.8 are computationally equivalent with respect to the final value of the relevant variable `z`. *There are two examples for a reordering*

Specification

```
rf[adrA]←a;
rf[adrB]←b;
mem[adr1]←val;
x←mem[adr2];
z←x+rf[adrR];
```

Implementation

```
(rf[adrB]←b,
 x←mem[adr2]);
(if adrA≠adrB
     then rf[adrA]←a,
 mem[adr1]←val);
(if adr1=adr2
     then z←val+rf[adrR]
     else  z←x+rf[adrR]);
```

Fig. 5.8: Examples for equivalent memory operations

of memory operations in Fig. 5.8. First, the order of the `read`*- and the* `store`*-operation to* `mem` *is reversed in the implementation. Thus,* `val` *is forwarded if the addresses are identical, otherwise the value assigned to* `x` *is used. This is a typical forwarding example occurring in pipelined systems. Second, the order of*

the `store`*-operations to the register file* `rf` *is reversed. This may, for example, occur during synthesis of architectures using data memory mapping, i.e., some single registers can be addressed by instructions in the same manner as registers of the register file. This is common for many microcontrollers, e.g., Microchip PIC or Intel 8051. Synthesis may change the order of accesses to this "common" data memory, e.g., by introducing pipelining. Formal verification has to consider the access to registers and register file by a single memory model. Otherwise it may remain unrevealed that, for example, the program counter is erroneously overwritten by an instruction due to a lacking address comparison.*

The memory model used by the symbolic simulator assumes an unlimited, but finite size for each memory in the descriptions. Memory access is modeled by the two array operations `read` and `store`. A new *RegVal* (for memories) with an incremented index is introduced after each `store`-operation to a memory. Only accesses to arrays that can be addressed by registers and not only by constants are considered by the `read`/`store`-model. Checking computational equivalence consists of comparing the respective final *RegVals* of the memories. The memory model, the indexing, equivalence of memory operations, and consideration of arrays addressed by constants are discussed in section 4.1.5.
Three types of equivalences have to be detected concerning memory-operations:

- **Value stored by a `store` is equivalent to a `read`** Section 5.9.2
  A `read`-operation reads for any acceptable initialization a value previously stored by a unique `store`-operation. Note that the `read`-operation occurs after the `store`-operation during simulation, i.e., this equivalence is only checked for `read`-operations.

- **Equivalence of two `read`-operations** Section 5.9.2
  Two `read`-operations are equivalent since they yield the same value for any acceptable initialization.

- **Equivalence of two `store`-operations** Section 5.9.3
  The resulting memory states are equivalent, i.e., the contents of the memories after the two `store`-operations in the specification and in the implementation are in any case identical. Often, the memory states *before* the `store`-operations are also equivalent, which is fast to check. The `stores` can also result in identical memory states in the opposite case for two reasons:

  – a `store`-operation is overwritten by subsequent `stores`;
  – the order of `store`-operations to the memory is different in the specification and in the implementation.

  Our equivalence detection is hierarchical: first an identical `store`-order in both descriptions is assumed, i.e., the memory states are pairwise identi-

cal. Then possibly overwritten `stores` are considered. Only if a `store`-operation has still no equivalent counterpart in the other description *and* a fast pre-check is satisfied, the more time consuming technique presented in the last part of section 5.9.3 is used to detect a changed order of `store`-operations.

As described in the previous sections, equivalence of terms is often decided by simply testing if the arguments are $\cong_{\mathcal{C}}$ or $\not\cong_{\mathcal{C}}$ which avoids the expansion of the arguments. This is also consequently used for the equivalence detection of `read`- and `store`-operations. Only the information of the *EqvClasses* of the addresses is used, i.e., our address comparison checks if two addresses $adr1$ and $adr2$ are

i. in the same *EqvClass*, i.e., $adr1 \cong_{\mathcal{C}} adr2$

ii. are in inequivalent *EqvClasses* $adr1 \not\cong_{\mathcal{C}} adr2$, or

iii. if equivalence depends on the initial register or memory values.

Expansion of arguments as in [VB98], where Boolean expressions are evaluated, is avoided. The following abbreviations are used in the examples of the next sections 5.9.2 and 5.9.3:

- Only the relevant `read`- and `store`-operations and address relations are shown. The generally complex control structure (e.g., *if-then-else*-clauses) and all assignments to registers which do not include a `read`-operation are omitted. Therefore, always only *one path* of the symbolic simulation is considered. Note that our equivalence detection for memory operations does *not* require additional case-splits.

- It is assumed that equivalences/inequivalences of the addresses have either already been determined by the other equivalence detection techniques described in the other sections of this chapter and chapter 6; or they are caused by case-splits at preceding conditions of *if-then-else*-clauses which are omitted, see above.

- Addresses or values with identical name in the specification and in the implementation, i.e., without the upper index s or i stand for arbitrary terms, which are assumed to have previously been detected $\cong_{\mathcal{C}}$. Using $adr1$ can signify textually different terms in both descriptions, e.g., $adr1^s_4 = \mathtt{a}^s_3 + \mathtt{b}^s_2$ and $adr1^i_3 = \mathtt{c}^i_1 + \mathtt{a}^i_2$, which are equivalent if $\mathtt{b}^s_2 \cong_{\mathcal{C}} \mathtt{c}^i_1$ and $\mathtt{a}^s_3 \cong_{\mathcal{C}} \mathtt{a}^i_2$ holds.

- The boxes below the examples indicate which additional relationships of the addresses must hold for two terms or memory states to be equivalent.

5.9.2 Detecting Equivalences of `Read`-Operations

Reading a Previously Stored Value

If the address of a `read`-operation reading from a memory and the address of the last `store`-operation referring to this memory are $\cong_{\mathcal{C}}$, then the value stored by this `store`-operation is always read.

Example 5.12
The memory state $\mathtt{mem}_1$ *in Fig. 5.9 (a) resulting from the last* `store`*-operation is the same as the first argument of the* `read`*-operation, i.e., the value stored is equivalent if the addresses are* $\cong_{\mathcal{C}}$.

This relationship does not hold if there is another intervening `store`*-operation as in Fig. 5.9 (b), since the second* `store`*-operation can overwrite the value stored by the first. But if the address of the* `read`*-operation is* $\not\cong_{\mathcal{C}}$ *to the address of the second* `store`*, its value is* in no case *read by this* `read`*-operation. For the* `read` *it seems as if the last* `store` *was not executed.*

(a) $\mathrm{mem}_1 \leftarrow \mathrm{store}(\mathrm{mem}, adr1, val1);$
$\mathrm{reg}_1 \leftarrow \mathrm{read}(\mathrm{mem}_1, adrR);$
$adr1 \cong_{\mathcal{C}} adrR$
$\Rightarrow \mathrm{reg}_1 \cong_{\mathcal{C}} val1$

(b) $\mathrm{mem}_1 \leftarrow \mathrm{store}(\mathrm{mem}, adr1, val1);$
$\mathrm{mem}_2 \leftarrow \mathrm{store}(\mathrm{mem}_1, adr2, val2);$
$\mathrm{reg}_1 \leftarrow \mathrm{read}(\mathrm{mem}_2, adrR);$
$(adr2 \not\cong_{\mathcal{C}} adrR) \wedge (adr1 \cong_{\mathcal{C}} adrR)$
$\Rightarrow \mathrm{reg}_1 \cong_{\mathcal{C}} val1$

Fig. 5.9: Reading previously stored values

In general, all preceding `stores` of a `read` with inequivalent addresses have to be ignored. This is done by calculating the *read access* of a `read`-operation, i.e., the relevant memory state. The addresses of all `store`-operations in between this memory state and the `read`-operation are inequivalent to the address of the `read`. The `store` previous to the *read access* has an address that is not inequivalent and its value might be read. If the address of this `store` is even $\cong_{\mathcal{C}}$, then the stored value is read in any case and, therefore, $\cong_{\mathcal{C}}$ to the `read`-operation.

Definition 5.2 (Read access)
Let $\mathcal{S} = \{\mathtt{store(mem_0, adr_0, val_0)}, \cdots, \mathtt{store(mem_x, adr_x, val_x)}\}$
be the `store`*-operations ordered by occurrence on the path previous to a* $read_R$*-operation with address* adr_R*.* $\mathcal{M}$ *denotes the corresponding series of memory states* mem_j *previous to the* `store`*-operations in* $\mathcal{S}$*. A* $store_j$ *has the address* adr_j *and the previous memory state* mem_j*. The read access of* $read_R$ *is*

$$read\_access(read_R) = mem_k \in \mathcal{M} : (\forall store_l \in \mathcal{S} \,|\, l \geq k \,:\, adr_R \not\cong_{\mathcal{C}} adr_l) \wedge (k = 0 \vee not\,(adr_{k-1} \not\cong_{\mathcal{C}} adr_R))$$

Note that the initial memory state is $mem_{k=0}$.

Equivalence of Read-Operations

Two `read`-operations from the specification and the implementation are equivalent if their addresses and their *read accesses* are equivalent. The equivalence of the *read accesses* guarantees that all locations of the memory where they might read from (depending on the actual value of the symbolic address) are identical.

Example 5.13
This procedure fails in the example of Fig. 5.10 if $adr1$ *is neither* $\not\cong_{\mathcal{C}}$ *nor* $\cong_{\mathcal{C}}$ *to* $adrR$*. The first* `store` *in the implementation is not relevant for the* `read`*-operation, if its address* $adrX$ *is inequivalent to* $adrR$*. But the read accesses of the two* `read`*-operations are not identical because of the intervening second* `store` *with* $adr1$*. Note that if* $adr1 \not\cong_{\mathcal{C}} adrR$ *holds, the read accesses would be both* `mem` *and if* $adr1 \cong_{\mathcal{C}} adrR$ *holds,* `val1` *would be read in both cases.*

Specification	**Implementation**
$\texttt{mem}_1^s \leftarrow \texttt{store}(\texttt{mem}, adr1, val1);$	$\texttt{mem}_1^i \leftarrow \texttt{store}(\texttt{mem}, adrX, valX);$
$\texttt{reg}_1^s \leftarrow \texttt{read}(\texttt{mem}_1^s, adrR);$	$\texttt{mem}_2^i \leftarrow \texttt{store}(\texttt{mem}_1^i, adr1, val1);$
	$\texttt{reg}_1^i \leftarrow \texttt{read}(\texttt{mem}_2^i, adrR);$

$$adrR \not\cong_{\mathcal{C}} adrX \Rightarrow \texttt{reg}_1^s \cong_{\mathcal{C}} \texttt{reg}_1^i$$

$adr1$ is assumed to be neither $\not\cong_{\mathcal{C}}$ nor $\cong_{\mathcal{C}}$ to $adrR$

Fig. 5.10: Equivalence of two `read`-operations

A supplementary check for two `read`-operations with equivalent addresses is provided to cope with mismatching *read accesses*. If the stored value and the address of the "intervening" `store`-operations are equivalent, then the *read access* is calculated again for both `read`-operations *without* these `stores`. This process can be repeated until either equivalent *read accesses* are found, i.e., the `read`-operations are equivalent, or intervening `store`-operations are reached that have not equivalent addresses/stored values. Note that the memory states of the intervening `store`-operations do not need be equivalent, see the example in Fig. 5.10.

Re-Checking Read-Operations

Our equivalence detection considers that the equivalence of the arguments of two terms is in most of the cases already obvious, when the second term is found on the path, see section 4.2. Therefore, it is sufficient to check only at the first occurrence of a term whether it is equivalent to some previously found term.

Frequently, not all equivalences and inequivalences concerning the addresses are already stated when finding `read`-operations for the first time on a path. This is common for memory operations since often a value is *speculatively* read or stored. An address conflict is checked afterwards to decide whether the speculation failed or not.

Example 5.14
The value of x is forwarded in Fig. 5.8 if there is an address conflict. If there is no conflict, equivalence of the `read`*-operation in the specification and in the implementation is only obvious after the case-split setting $adr1 \not\cong_C adr2$.*

Decisions about addresses later on a path as in Example 5.14 are frequent for processor designs with pipelining. A value is read speculatively and used only if there is no data conflict. Otherwise the relevant value is forwarded. The example indicates, that it is important to check `read`-operations whenever the *EqvClasses* of the corresponding addresses are modified. Therefore, the `read`-operations found during symbolic simulation on a path are marked at the *EqvClasses* of their addresses as *dependent* `read`-operations. If there is a change of an *EqvClass*, either because it is unified or set inequivalent to another *EqvClass*, all *dependent* `read`-operations are checked again, see also section 4.3. In the example of Fig. 5.8, the `read`-operation in the specification is marked at the *EqvClass* of `adr2`. The equivalence of the `read`-operations is detected, when setting the *EqvClasses* of `adr1` and `adr2` inequivalent.

5.9.3 Detecting Equivalent Memory States

Detecting the equivalence of two memory states is necessary to demonstrate computational equivalence but also required to argue about the equivalence of two `read`-operations in the specification and in the implementation. Finding equivalent memory states is the same as detecting equivalent `store`-operations, since a `store`-operation returns the whole new memory state.

Identical Order of Store-Operations

For some designs, the order of `store`-operations is identical in the two descriptions to be compared. A sufficient, but not necessary condition for the equivalence of two `store`-operations and, therefore, the resulting memory states is that the addresses, the values stored, and the *previous* memory states are pairwise in the same *EqvClass*. This is fast to test and, therefore, checked first when finding a new `store`-operation. The final values of a memory in the implementation and in the specification depend on the last two `stores` on both sides, which use the result of the previous `stores` as first arguments. By means of an inductive argument, when building a list in order of appearance of the `stores` in the implementation and in the specification, every `store` may have its "partner" on the other side, if the order of `store`-operations is identical. The first `store`-operations on both sides have the initial memory state as first argument, which is identical.

The specification and the implementation can have also only *partially* identical orders of `stores`, which begin from two equivalent memory states. These states may be either the initial memory state or memory states that have been iden-

tified to be equivalent by one of the techniques described below. The partially identical `store`-order ends before the first `store`-operation-pair, where either the addresses or the stored values are not equivalent.

Definition 5.3 (Identical `store`-order)
Let $\mathcal{S}_{spec} = \{\texttt{store}(\texttt{mem}^s_x,\texttt{adr}^s_x,\texttt{val}^s_x),\cdots,\texttt{store}(\texttt{mem}^s_{x+n},\texttt{adr}^s_{x+n},\texttt{val}^s_{x+n})\}$
$\mathcal{S}_{impl} = \{\texttt{store}(\texttt{mem}^i_y,\texttt{adr}^i_y,\texttt{val}^i_y),\cdots,\texttt{store}(\texttt{mem}^i_{y+n},\texttt{adr}^i_{y+n},\texttt{val}^i_{y+n})\}$
be `store`*-operations in the specification and in the implementation ordered by occurrence on the path. An identical* `store`*-order satisfies:*

$$\forall k = 0,\ldots,n : \texttt{adr}^s_{x+k} \cong_{\mathcal{C}} \texttt{adr}^i_{y+k} \wedge \texttt{val}^s_{x+k} \cong_{\mathcal{C}} \texttt{val}^i_{y+k}$$

If the memory states previous to the `store`*-operations (*$\texttt{mem}^s_x$ *and* $\texttt{mem}^i_y$*) are equivalent then the same holds for the resulting memory states, i.e.,* $\texttt{mem}^s_{x+n+1} \cong_{\mathcal{C}} \texttt{mem}^i_{y+n+1}$.[15]

The order of `store`-operations has to be the same in the specification and in the implementation only with regard to the same specific memory. The interleaving of `store`-operations to different memories can be arbitrary, an example is given in Fig. 5.11.

```
store(dmem,adr1,val1)                    store(dmem,adr1,val1)
store(rf,adr2,val2)      have the same   store(dmem,adr3,val3)
store(dmem,adr3,val3)    store order for store(rf,adr2,val2)
store(rf,adr4,val4)      both rf and dmem store(rf,adr4,val4)
```

Fig. 5.11: Identical `store`-orders

Overwritten Store-Operations

An identical `store`-order requires an equal number of `store`-operations on the current path, which is not a necessary condition for equivalence of the resulting memory states.

Example 5.15
An additional store occurs in the implementation of Fig. 5.12. Nevertheless, the final memory states are identical if the value stored by the second `store`*-operation of the implementation is in any case overwritten by the third* `store`*-operation, i.e., if the addresses are* $\cong_{\mathcal{C}}$.

This situation can occur, for instance, if the second `store` is speculative, but speculation fails and the third `store` is used to correct the fault. Let us assume that *val*2 and *valX* are not $\cong_{\mathcal{C}}$. Therefore, $\texttt{mem}^s_2$ and $\texttt{mem}^i_2$ cannot be in the same *EqvClass* and the equivalence detection of the previous subsection will fail. But there is no difference for the last `store`-operation in the implementation if the previous memory state is $\texttt{mem}^i_1$ or $\texttt{mem}^i_2$. Therefore the *relevant preceding*

[15] This not a necessary condition, see below.

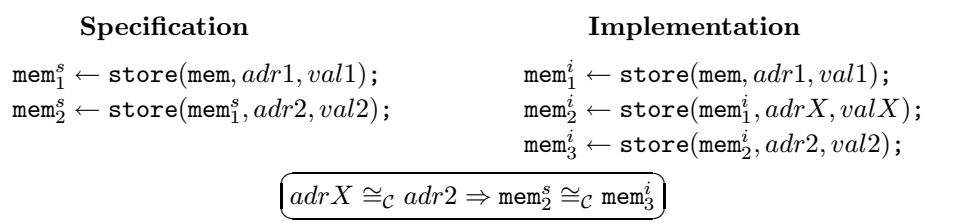


Fig. 5.12: Example for an overwritten store-operation

memory state is calculated for equivalence checking. This is either the memory state after the first preceding `store`-operation, which is not overwritten by the new `store`-operation or the initial memory state.

Definition 5.4 (Relevant preceding memory state)
Let $\mathcal{S}_{spec} = \{\mathtt{store(mem_0,adr_0,val_0)},\cdots,\mathtt{store(mem_{x-1},adr_{x-1},val_{x-1})}\}$ *be the* `store`*-operations previous to* $store_x$ *with the address* adr_x *and the value* val_x*.* $\mathcal{M}$ *denotes the corresponding series of memory states previous to the* `store`*-operations in* $\mathcal{S}$*. Note that* $\mathtt{adr}_i$ *and* $\mathtt{val}_i$ *stand for arbitrary terms, see section 5.9.1. The initial memory state is* $\mathtt{mem}_0$*.*
The relevant preceding memory state of $store_x$ *is*

$$rel\_prec\_state(store_x) = mem_k \in \mathcal{M} : (\forall store_l \in \mathcal{S} \mid l \geq k \,:\, adr_x \cong_{\mathcal{C}} adr_l) \wedge (k = 0 \vee not\,(adr_{k-1} \cong_{\mathcal{C}} adr_x))$$

Two `store`-operations in the specification and in the implementation are equivalent if the addresses, the stored values and the *relevant preceding memory states* are $\cong_{\mathcal{C}}$. This criterion copes with different number of overwritten `store`-operations in the specification and in the implementation. Determining the *relevant preceding memory state* is fast, since, again, only the information of the *EqvClasses* is used. Furthermore, its calculation is only necessary if there exists a potential "counterpart" with equivalent address and stored value.

Note that by considering overwritten `stores`, there are some special cases where more than two `store`-operations - one of the specification and one of the implementation - are in a single *EqvClass*. For instance, the memory states after the second and the third `store` in the implementation in Fig. 5.12 are identical if $adr2 \cong_{\mathcal{C}} adrX$ and $val2 \cong_{\mathcal{C}} valX$ hold.

Changed Order of Store-Operations

If the `store`-order is changed as in the example of Fig. 5.8 for `rf` and Fig. 5.13 for `mem`, then the final memory states can be equivalent, if the addresses of the `store`-operations are $\ncong_{\mathcal{C}}$. A correct reordering of `store`-operations can be the result, for example, of synthesizing designs with data mapping, see section 5.9.1.

When a new `store`-operation is found and all previous checks fail, there might exist a `store` in the other description with equivalent address and stored value, which is the "counterpart" in a changed `store` order. Since the new `store` is

the most recent in its description, there must be some `store`-operations *before* it, which happen *after* the "counterpart" in the other description.

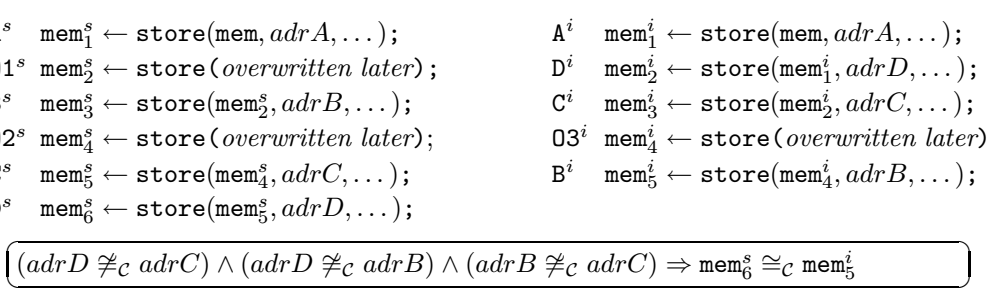


Fig. 5.13: Changed order of `store`-operations

Assume that the new `store` is $\texttt{D}^s$ and the "counterpart" $\texttt{D}^i$ in Fig. 5.13. The stores `B` and `C` are before `D` in the specification but after `D` in the implementation. The `stores O1, O2, O3` are overwritten by subsequent `store`-operations, i.e., $\texttt{B}^s$, $\texttt{C}^s$, $\texttt{D}^s$, or $\texttt{B}^i$. A valid reordering of the `store`-operations requires that the addresses of `D` on the one hand and `B`, `C` on the other hand are $\not\cong_{\mathcal{C}}$. But we do not know that only `B` and `C` have to be checked, since there might be some overwritten stores $\texttt{O1}^s$, $\texttt{O2}^s$, or $\texttt{O3}^i$ in between or before `B` or `C` (see Fig. 5.13). For a quick test, first two sets containing all memory states *previous* to $\texttt{D}^s/\texttt{D}^i$ are determined, where all `store`-operations after those memory states and before $\texttt{D}^s/\texttt{D}^i$ have a determined address relationship; i.e., the addresses of those `store`-operations must be either $\not\cong_{\mathcal{C}}$ to the address of $\texttt{D}^s/\texttt{D}^i$ or $\cong_{\mathcal{C}}$ to the address of one of the subsequent `store`-operations. A changed `store`-order is only checked, if there are equivalent memory states in those two sets calculated for $\texttt{D}^s$ and $\texttt{D}^i$. In the following, this is called that $\texttt{D}^s$ and $\texttt{D}^i$ have a *common access state*, a formal definition is given on page 93.

The next step is to determine the two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ containing the same `store`-operations appearing in the two descriptions in changed order. This is not obvious since only the end of $\mathcal{S}_1$ and the beginning of $\mathcal{S}_2$ are known. Furthermore, overwritten stores have to be considered correctly, i.e., $\mathcal{S}_1 = \{\texttt{B}^s, \texttt{C}^s, \texttt{D}^s\}$ `and` $\mathcal{S}_2 = \{\texttt{D}^i, \texttt{C}^i, \texttt{B}^i\}$ in Fig. 5.13. We assume in the following that all `store`-operations of the changed `store`-order have already appeared first in the implementation ($\texttt{D}^i$ to $\texttt{B}^i$) and now the last `store` of the opposite sequence $\texttt{store}_{end}^{\mathcal{S}_1}$ is detected during the simulation, i.e., $\texttt{D}^s$. This is the first time where again equivalent memory states can be reached. Since $\texttt{D}^s$ is the most recent `store` detected during simulation, the algorithm assumes that this is the last element missing and that it is the end of $\mathcal{S}_1$. Tracing back from this point, the first (previous) memory state is searched, which has an equivalent counterpart in the other description, i.e., $\texttt{mem}_1^s$ and $\texttt{mem}_1^i$ in Fig. 5.13. All preceding `stores` do not have to be considered since they lead to an equivalent memory state in the implementation and in the specification. The `store`-operations in the two descriptions directly

after this equivalent memory state $\texttt{store}^{\mathcal{S}_1}_{begin}$ ($\texttt{O1}^s$) and $\texttt{store}^{\mathcal{S}_2}_{begin}$ ($\texttt{D}^i$) are the beginnings of $\mathcal{S}_1$ and $\mathcal{S}_2$ before eliminating overwritten `store`-operations.

Overwritten `stores` can be removed easily in $\mathcal{S}_1$ since the latest $\texttt{store}^{\mathcal{S}_1}_{end}$ ($\texttt{D}^s$) is known. Tracing back from $\texttt{store}^{\mathcal{S}_1}_{end}$ ($\texttt{D}^s$) to $\texttt{store}^{\mathcal{S}_1}_{begin}$ ($\texttt{O1}^s$), all `store`-operations with an address which is $\cong_{\mathcal{C}}$ to the address of a subsequent `store` are eliminated, which results in $\mathcal{S}_1 = \{\texttt{B}^s, \texttt{C}^s, \texttt{D}^s\}$.

The end $\texttt{store}^{\mathcal{S}_2}_{end}$ of the sequence $\mathcal{S}_2$ is unknown, which makes eliminating overwritten `store`-operations harder. Symbolic simulation may have already reached some `store`-operation *after* $\texttt{B}^i$ which overwrites, for instance, $\texttt{C}^i$ but has to be ignored to determine $\mathcal{S}_2$ correctly. All `store`-operations *after* the unknown final $\texttt{store}^{\mathcal{S}_2}_{end}$ ($\texttt{B}^i$) do not have to be considered when eliminating overwritten `stores` in $\mathcal{S}_2$. $\mathcal{S}_2$ is determined by beginning with $\texttt{store}^{\mathcal{S}_2}_{begin}$ and adding successively subsequent `stores`. Every time a new `store` is added, possibly overwritten `stores` are eliminated. This process is stopped, when the number of `store`-operations in $\mathcal{S}_2$ is the same as in $\mathcal{S}_1$.

Finally, it is controlled, if every `store`-operation in $\mathcal{S}_1$ has its partner in $\mathcal{S}_2$ with $\cong_{\mathcal{C}}$ address, $\cong_{\mathcal{C}}$ stored value and *common access state* (see above and Definition 5.5 below). In this case, the memory states after $\texttt{store}^{\mathcal{S}_1}_{end}$ and $\texttt{store}^{\mathcal{S}_2}_{end}$, i.e., $\texttt{D}^s$ and $\texttt{B}^i$ are equivalent. Note that the technique described in this section is not limited with respect to the length of the changed `store` order, which is three in our example.

The handling of some exceptional situations is not discussed in this work for brevity. Consider for example that a $\texttt{store}\ \texttt{E}^i$ follows directly $\texttt{B}^i$, which overwrites $\texttt{C}^i$ with exactly the same value as $\texttt{C}^i$. $\texttt{D}^s$ is then not only equivalent to $\texttt{B}^i$ but also to $\texttt{E}^i$. This is detected by building two sequences $\mathcal{S}_{2a}$ and $\mathcal{S}_{2b}$ with $\texttt{B}^i$ and $\texttt{E}^i$ as last elements in this special case.

The following definition gives the conditions of a valid changed `store`-order. Note that the identification of such an order by the symbolic simulator as described by the previous example is optimized with respect to computation time.

Definition 5.5 (common access state, valid changed store order)
Let

$$\mathcal{S}^{w/o\ overwrit}_{spec} = \{\texttt{store}(\texttt{mem}^s_x, \texttt{adr}^s_x, \texttt{val}^s_x), \cdots, \texttt{store}(\texttt{mem}^s_{x+n}, \texttt{adr}^s_{x+n}, \texttt{val}^s_{x+n})\}$$
$$\mathcal{S}^{w/o\ overwrit}_{impl} = \{\texttt{store}(\texttt{mem}^i_y, \texttt{adr}^i_y, \texttt{val}^i_y), \cdots, \texttt{store}(\texttt{mem}^i_{y+n}, \texttt{adr}^i_{y+n}, \texttt{val}^i_{y+n})\}$$

be `store`*-operations in the specification and in the implementation ordered by occurrence on the path. All overwritten* `store`*-operations are previously eliminated, i.e.,*

$$\mathcal{S}^{w/o\ overwrit}_{spec} = remove\_overwritten(\mathcal{S}_{impl})$$
$$\mathcal{S}^{w/o\ overwrit}_{impl} = remove\_overwritten(\mathcal{S}_{spec})$$
$$remove\_overwritten(\mathcal{S}) = \{store_k \in \mathcal{S} : \nexists store_l \in \mathcal{S} \mid l > k\ :\ adr_k \cong_{\mathcal{C}} adr_l\}$$

$\mathcal{M}$ denotes the corresponding series of memory states previous to the `store`*-operations in a series $\mathcal{S}$. Let mem_j, adr_j, and val_j be the previous memory state, the address, and the value of $store_j$. The set of access states of $store_z$ in an order of* `store`*-operations $\mathcal{S}$ is:*

$$access(store_z) = \{mem_k \in \mathcal{M} : \forall store_l \in \mathcal{S} \mid z > l \geq k : adr_l \cong_{\mathcal{C}} adr_z \vee adr_l \ncong_{\mathcal{C}} adr_z\}$$

Two `store`*-operations of the specification $store^s_m$ and of the implementation $store^i_n$ have a common access state if:*

$$common\_access(store^s_m, store^i_n) = \exists mem^s_k \in access(store^s_m),\ mem^i_l \in access(store^i_n) :\ mem^s_k \cong_{\mathcal{C}} mem^i_l$$

Note that mem_j denotes the memory state previous to a $store_j$. If the store order of $\mathcal{S}^{w/o\ overwrit}_{spec}$ and $\mathcal{S}^{w/o\ overwrit}_{impl}$ are not identical according to Definition 5.3 then a valid changed store order is given if:

$$\forall store^s_k \in \mathcal{S}^{w/o\ overwrit}_{spec} : \exists store^i_l \in \mathcal{S}^{w/o\ overwrit}_{impl} : (adr^s_k \cong_{\mathcal{C}} adr^i_l) \wedge (val^s_k \cong_{\mathcal{C}} val^i_l) \wedge common\_access(store^s_k, store^i_l)$$

If the memory states previous to the `store`*-operations*[16] *are equivalent then the same holds for the resulting memory state, i.e.,* $\mathtt{mem}^s_{x+n+1} \cong_{\mathcal{C}} \mathtt{mem}^i_{y+n+1}$.

5.9.4 Summary

Symbolic simulation has to cope with two aspects concerning memories: first, the in general large sizes of the memories. We argue only about memory operations, i.e., `store`- and `read`-operations. Therefore, the size of the memories is irrelevant, but the symbolic simulator has to detect equivalences of the memory operations in order to model correctly the behavior of the memory.

Second, indirect addressing has to be considered. This makes necessary a reasoning process about the relationships of the addresses during symbolic simulation, since they can be arbitrary symbolic terms. Collecting equivalent symbolic terms in *EqvClasses* permits us to establish a fast address comparison for our memory-specific equivalence detection methods. The symbolic simulator copes with complex reorderings of memory operations as demonstrated also by the experimental results presented in section 7.1.

[16] Which need not be $\mathtt{mem}^s_x$ and $\mathtt{mem}^i_y$ since the first `store`-operations might be overwritten, i.e., $\mathcal{S}_{spec/impl}$ are relevant instead of $\mathcal{S}^{w/o\ overwrit}_{spec/impl}$.

5.10 Inequivalences Forcing Terms to be Constant

Inequivalences can force a term to be constant. Since the domain of a n-bit-vector is restricted to 2^n values, setting it $\not\cong_{\mathcal{C}}$ to $2^n - 1$ values implies equivalence to the remaining value. Fig. 5.14 (a) gives an example for a bit-vector, where `b`$\not\cong_{\mathcal{C}}$`10 and b`$\not\cong_{\mathcal{C}}$`00 and b`$\not\cong_{\mathcal{C}}$`11` $\Rightarrow$ `b` $\cong_{\mathcal{C}}$`01` holds. Note that there can be intervening assignments and other conditions in Fig. 5.14.

```
(a) if b="10" then ...
    elsif b="00" then ...
    elsif b="11" then ...
    else [ b="01" is true ]
```

```
(b) if a[3] then
        if a[2:1]="11" then ...
        elsif a[2:1]="10" then ...
        else [ a[3:2]="10" is true ]
```

Fig. 5.14: Terms being constant due to decided inequivalences

Two *EqvClasses* are inequivalent either because of a decision in a case split or since they contain different constants which is not relevant here. Checking after each decision whether the *EqvClass* is set inequivalent to $2^n - 1$ constants is not sufficient. Also decisions about *parts* of a term have to be considered, see Fig. 5.14 (b) where `a[3]`$\cong_{\mathcal{C}}$`1 and a[2:1]`$\not\cong_{\mathcal{C}}$`11 and a[2:1]`$\not\cong_{\mathcal{C}}$`10` $\Rightarrow$ `a[3:2]`$\cong_{\mathcal{C}}$`10` has to be detected. Moreover, it is not relevant in this example whether the entire term `a` is equivalent to a constant but only the bits `a[3:2]`.

Two counters *ctrl-zero-bit* and *ctrl-one-bit* are introduced for each bit of a term appearing in conditions. They are initialized during pre-processing with 2^{N-1} where N is the length of the term (not $2^N - 1$!). A bit i of a term is equivalent to 0 (1) if *ctrl-zero-bit*$_i$ (*ctrl-one-bit*$_i$) is zero. The counters are decremented if:

- the term is set inequivalent to a constant. *ctrl-one-bit* and *ctrl-zero-bit* are decremented at all bit-positions, where this constant is 0 or 1, respectively; a table supports determining the relevant bit-positions for constants appearing explicitly in the descriptions since all constants are expressed as integers during simulation, see section 4.3;

- a *bit-selection* of a term is set inequivalent to a constant. Not only the *ctrl-one-bit-* and *ctrl-zero-bit*-counters of the term representing the bit-selection, e.g., `a[2:1]` are decremented but also the corresponding counters of the entire term `a` are decremented according to the size of the bit-selection. Multiple selections, e.g., `(a[10:2])[2:1]` are considered by recursion.

Every time a new constant bit is found it is checked whether the whole term is constant, too. The equivalence of the bit has to be marked if this test fails. If there exists a term representing the *bit-selection* of the relevant bit then the corresponding *EqvClasses* are unified. Otherwise equivalence of the bit to the constant 0 or 1 is marked directly at the term.

Chapter 6

Using Decision Diagrams to Detect Equivalences

Section 6.1 gives an overview of the *dd-checks*. The construction of formulas which demonstrate the equivalence to be verified is described in section 6.2. Checking those formulas by vectors of *OBDDs* is compared to other techniques in section 6.3. The use of intermediate *dd-checks* for gate-level simulation is presented in section 6.4. Section 6.5 discusses how the decisions of conditions are considered during a *dd-check*. The results of a *dd-check* are reused during the following symbolic simulation of the remaining paths which is described in section 6.6.

6.1 Overview

The equivalence detection techniques presented in the previous chapter are not complete in order to provide a fast symbolic simulation. Therefore, checking the verification goal by a test for equivalence at the end of a path (line 11 in Algorithm 4.1) may fail. The more accurate tests called *dd-checks* based on decision diagrams are used at the end of a path in these cases. They have to reveal whether (i) computational equivalence is given in this path but was not detected (line 16), (ii) a condition has been decided inconsistently due to the incomplete equivalence detection on the fly (line 19), or (iii) a valid counterexample can be given (line 22).

Decision diagrams are used in the *dd-checks* to reveal special equivalences which are not considered by the techniques presented in the previous chapter either since they occur seldom or because they are hard to detect. Examples are given in section 6.4, 6.5, and 7.3. Two tests are provided:

- testing whether two terms are equivalent; note that checking the validity of a condition is the same as comparing it to the constant 1;
- testing whether a term is equivalent to a constant; this is a different case since the value of the constant is unknown.

A formula demonstrating the equivalence is built for each test considering knowledge about path-dependent equivalences or inequivalences of intervenient terms. The *Multiple-Domain Decision Diagram Package* (the TUDD-package) [Hör99, Hör97, Hör98] developed at Darmstadt University of Technology with an extension for vectors of *OBDDs* is used to prove the formula. Each graph represents one bit of the two terms to be compared. The extension developed for the symbolic simulator permits to apply functions to vectors of *OBDDs* instead of manipulating separately single decision diagrams.[1] Therefore, the formula consisting of function applications to bit-vectors is checked automatically by the TUDD-package without additional modifications. It is tested whether a similar formula has been built previously and stored in a hash-table before applying vectors of *OBDDs*.

The *dd-checks* testing the verification goal at the end of the path may fail if a false path is reached. All decided conditions (i.e., *CondBits* in $\mathcal{C}$ for which a case-split was performed) are checked in order of their occurrence in this case to search for a contradictory decision due to the incomplete equivalence detection on the fly. Using the information of the equivalence classes again facilitates considerably the construction of the required formulas.

A path is backtracked if at least one formula is valid (line 16 in Algorithm 4.1) or if a contradictory decision has been detected (line 19). Moreover, the relationship revealed by the *dd-check* is marked as described in section 6.5 so that it is checked during symbolic simulation of the remaining paths. Otherwise a valid counterexample is found which is reported for debugging.

Section 4.6 motivated the use of intermediate *dd-checks* at gate-level also *during* the path search (line 9 in Algorithm 4.2) instead of using them only at the end of a path. The intermediate tests are discussed in section 6.4.

The *dd-checks* do not make the equivalence detection complete, since some functions like multiplication or memory operations are not interpreted during the *dd-check* to avoid extensive computation times and/or graph explosion. These terms are represented by *dd-cutpoints* (described in the next section) during *OBDD*-construction. Additional *dd-cutpoints* are used to speed up the *dd-checks*. In spite of these simplifications, the *dd-checks* provide a substantial improvement of the equivalence detection. No corner-case has been found during our experiments which was not detected by the implemented *dd-checks*. Note that the *dd-checks* need not be incomplete *in principle* in our symbolic simulation approach. The incompleteness is caused by the *dd-cutpoints* which are only introduced because of the *practical* limitations of current *OBDD*-packages.

[1] For example, the application of the function ADD to two vectors of *OBDDs* $\{a_0, \ldots, a_n\}$ and $\{b_0, \ldots, b_n\}$ is implemented using the basic functions `and`, `or`, and `xor` of the TUDD-package. The result is another vector of *OBDDs*.

6.2 Building Formulas in *dd-checks*

The support of two terms has to be the same if the equivalence of the terms is tested using decision diagrams. This can be achieved by backward-substitution so that only initial *RegVals*, which are identical in the specification and in the implementation, or constants occur on each side. Note that the formula is less complex than a formula describing the entire verification problem since a specific path is chosen. However, a complete backward-substitution is not efficient since only the information about the path is used but not about equivalences detected by the other techniques. For example, if both terms depend only on two intermediate *RegVals* detected previously to be equivalent, it makes sense to introduce a *dd-cutpoint* and to consider this *dd-cutpoint* as primary input: all expressions or assignments previous to this *dd-cutpoint* do not have to be considered in the decision diagrams.

Therefore, first the two sets representing all *EqvClasses* of the intermediate terms are collected in a fast backtracking. The intersection of those two sets of *EqvClasses* represents the candidates for *dd-cutpoints*. Any term with an *EqvClass* in the intersection is represented by a *dd-cutpoint* when constructing the formula by backward-substitution, i.e., the *dd-check* considers this term as a primary input just as the initial *RegVals*.

The *dd-cutpoints* have to be removed in some cases since they hide subterms which are required to demonstrate equivalence, see section 6.4 for an example. Therefore, a failed *dd-check* is repeated without *dd-cutpoints*.

Another possibility to obtain a simpler formula is to replace a term during formula construction by another term in the same *EqvClass*. Again, the results of the previous symbolic simulation are used. Replacing a term by another term in the same *EqvClass* is useful if the corresponding representation as decision diagram is simpler. For example, if a term is in an *EqvClass* with a constant, then only the *OBDD* for the constant is constructed. A simple heuristic counts the expected complexity concerning graph construction of each term in the *EqvClass* of a term. The term with the lowest complexity is used as representative for the *EqvClass*, i.e., it replaces the other terms in the *dd-check*.

Replacing terms by other terms in the same *EqvClass* or by *dd-cutpoints* can be misleading if the consistency of the decided *CondBits* is verified, i.e., if it is checked whether a false path is reached (line 19 in Algorithm 4.1). If the condition of an inconsistent *CondBit* establishes an equivalence of two terms, then replacing the terms by a *dd-cutpoint* or one of the terms by the other term makes detecting the inconsistency infeasible. Therefore, all *EqvClasses* with a term appearing in the condition of a *subsequently* decided *CondBit* have to be ignored when checking the consistency of a decided *CondBit*. Terms in such a *EqvClass* are replaced neither by *dd-cutpoints* nor by other terms of this *EqvClass*.

A hash-table is used to avoid building identical decision graphs repeatedly in

different *dd-checks*. The result of a previous *dd-check* can be reused even if the two formulas of the *dd-checks* are not identical. The same formula may be built in the new *dd-check* with only different *RegVals* or *dd-cutpoints*. Therefore, all *RegVals* and *dd-cutpoints* are replaced in order of their appearance in the formula by auxiliary variables T1, T2,...,Tn before hashing a formula. New formulas are checked using vectors of *OBDDs* and the result is hashed.

Note that a verification using *only* vectors of *OBDDs* without considering results of the symbolic simulation is neither efficient nor feasible for large examples, see section 7.3. A small example for the simplification of a formula in a *dd-check* by using results of the other equivalence detection techniques is given in Fig. 4.14 in section 4.5.2.

6.3 Comparison to Other Approaches for Formula-Checking

A *dd-check* consists of extracting first a formula which is valid if the two terms to be compared are equivalent and then verifying this formula by means of vectors of *OBDDs*. The formula established could be verified also by other techniques. Two of them are compared in our domain of application to vectors of *OBDDs* in the following: another type of decision diagrams and a specialized formula checker called SVC, see section 3.3. Note that techniques which require possibly user-interaction to check a formula, e.g., theorem-provers are not suited for our automatic verification approach.

A different possibility to represent and check a formula is to use word-level decision diagrams like **BMDs* [BC94, BC95] instead of vectors of *OBDDs*. *Bit-selections* are used frequently in practical examples of control logic, either explicitly, e.g., `R[13:16]`, or implicitly, e.g., storing the result of an addition in a register without carry. Using **BMDs*, terms are represented by one single **BMD*. *Bit-selection*, therefore, requires one or two *modulo*-operations which are worst-case exponential with **BMDs*.

Bit-selection is quasi for free, if terms are expressed as *vectors* of *OBDDs*, where each graph represents one bit. *Bit-selection* can then be done by simply skipping the irrelevant bits, i.e., the corresponding *OBDDs* and by continuing computation with the remaining graphs. Checking equivalence just consists of comparing each bit-pair of the vectors.

All previously applied equivalence detection techniques are (fairly) independent of the bit-vector length. Results obtained during symbolic simulation are used to simplify formulas before *OBDD*-vector construction. But even without simplification, large bit-vectors can be handled by *OBDD*-vectors in acceptable computation time.

The results of SVC on five bit-vector arithmetic verification examples are compared in [BDL98] to the results of the **BMD* package from Bryant and Chen

and also to Laurent Arditi's **BMD* implementation which has special support for bit-vector and Boolean expressions. We verified these examples also with *OBDD*-vectors. Tab. 6.1 summarizes the results. All our measurements are on a Sun Ultra II with 300 MHz. Various orderings of the variables for our **BMD*-measurements are used; the best results are reported. The line DM contains additional verification results for a bit-wise application of De Morgan's law to two bit-vectors a and b, i.e., $\overline{a_0 \wedge b_0}\&...\&\overline{a_n \wedge b_n} \equiv (\overline{a_0} \vee \overline{b_0})\&...\&(\overline{a_n} \vee \overline{b_n})$, and the ADD-example is the verification of a ripple-carry-adder. Note that the input for the two last examples is also one *word* and not a vector of inputs. Otherwise **BMD*-verification is of course fast since no *bit-selection* or modulo operation is required. The inputs may represent some intermediate cut-points for which, e.g., the **BMD* is already computed.

	SVC[1]		*BMD Bryant/Chen[1]		*BMD Arditi[1]		OBDD-vector TUDD				
	200MHz Pentium		200MHz Pentium		300MHz UltraSparc 30		300MHz Sun Ultra II				
Bits	16	32	16	32	16	32	16	32	64	128	256
1	N/A	0.002	N/A	N/A	N/A	0.04	0.14	0.27	0.38	0.68	1.38
2	N/A	0.002	N/A	N/A	N/A	1.10	0.13	0.20	0.25	0.44	0.93
3	0.002	0.002	265.0	>500	0.07	0.18	0.21	0.32	0.51	0.95	1.95
4	0.002	0.002	26.4	>500	0.72	8.79	0.24	0.40	0.71	1.53	4.38
5	0.111	0.520	22.7	>500	0.39	3.78	0.14	0.21	0.31	0.57	1.15
	Measured at TUDD		Measured with TUDD *BMD-package								
Bits	16	32	16	32	64		16	32	64	128	256
DM	>5min		>5min				0.12	0.22	0.28	0.48	1.03
ADD	-[2]		5.19	37.2	282.7		0.21	0.31	0.48	0.98	1.90

[1] Measurements reported in [BDL98].
[2] 2 Bit: 1.01s; 4 Bit: 9.47s; 5 Bit 44.69s; Verification with more than 5 Bit was not feasible with the current version of SVC.

Tab. 6.1: Comparison of SVC, *BMD and OBDD-Vectors. Times are in seconds

Obviously, **BMD*-verification suffers from the modulo-operations in the examples. According to [BDL98], the results of example 1 to 4 are independent of the bit-vector length for SVC, but the verification times with *OBDD*-vectors are also acceptable even for large bit-vectors. These times can be reduced especially for small bit-vectors by optimizing our formula parsing. In example 5, SVC ends up slicing the vector. Thus the execution time depends on the number of bits and shows, therefore, a significant increase, whereas the computation time for *OBDD*-vectors increases only slightly. The increase in *this* example may be eliminated in a future version of SVC [BDL98], but the general problem is that slicing a vector has to be avoided in SVC. This is demonstrated by the examples DM and ADD, where verification is only practical with *OBDD*-vectors.

Note that functions that are worst-case exponential with *OBDDs*, e.g., multiplication or which have no representation are only problematic in rare cases where

special properties of the functions are necessary to show equivalence. Normally, these terms are replaced by *dd-cutpoints* during formula-construction since information from the other equivalence detection techniques is used.

6.4 Comparing Descriptions at RT- and Gate-Level

Section 4.6 motivated the use of intermediate *dd-checks during* the path search if one of the descriptions is at gate-level instead of using them only at the end of a path (line 9 in Algorithm 4.2). The same entire Boolean expressions assigned to the register bits have to be simulated at gate-level in each symbolic simulation cycle. It is crucial to find relationships of the *control* registers in the *previous* cycle in order to detect equivalences in the next cycle between the Boolean expressions at gate-level and the much simpler corresponding terms in the specification at algorithmic- or rt-level. Usually, the control registers appear frequently in the Boolean expressions. The equivalence detection techniques presented in section 5.2 can often neglect subterms or decide equivalences if information is provided about the value of the control registers.

Example 6.1
The register `cnt` *in Fig. 6.1 is assumed to be a microprogram counter and the assignments to all registers depend on the value of this control register. The assignment to* `cnt` *is represented at gate-level by a concatenation (&) of the single bits. Only the expression of one bit is shown in Fig. 6.1. This bit*

Specification	**Implementation**
if `ak[3:0]=mi[3:0]`	$\texttt{cnt}_1^i \leftarrow bit_n$ `&...&`
then ...	`((ak[3] xor mi[3]) nor (ak[2] xor mi[2])) and`
else *selected branch*;	`((ak[1] xor mi[1]) nor (ak[0] xor mi[0]))`
	`&...&` bit_0`;`

Fig. 6.1: Example for the advantages of intermediate *dd-checks*

is constant since `ak[3:0]`$\not\cong_C$`mi[3:0]` $\Rightarrow$ `((ak[3] xor mi[3])` $\cdots$`(ak[0] xor mi[0]))` $\cong_C$ `0` *which is not revealed without dd-check by the other equivalence detection techniques.*

Detecting that the (controlling) microprogram counter is equivalent to a constant is important, since the assignments to all registers in the next cycle are identified to be equivalent to a corresponding RegVal in the specification in this case. Otherwise the "link" between terms in the specification and in the implementation gets lost not only in the next cycle but also in all subsequent cycles since the respective preceding RegVals are used as arguments. A final dd-check would be complex since the subsequent cycles have to be considered in the decision diagrams additionally before equivalent terms of the specification and of the implementation are reached.

Losing the "link" is avoided by providing an intermediate *dd-check* at gate-level if no equivalence has been found for a term assigned to a *RegVal*. This intermediate *dd-check* reveals in Example 6.1 that $\texttt{cnt}_1^i$ is equivalent to 0.

An intermediate check is provided for a *RegVal* in the description at gate-level if the following conditions hold:

- no equivalence between the term assigned to the *RegVal* and any other term has been detected by the other equivalence detection techniques;
- the register is not excluded from intermediate *dd-checks*; the user can limit the application of intermediate *dd-checks* on relevant control registers; this (easily provided) information is optional, but can decrease simulation time significantly.

The *dd-check* requires an assumption about which term might be equivalent to the intermediate $RegVal_x^i$. If the register does not exist in the specification, e.g., a control register of the hardware implementation, it is only checked

- whether the term is equivalent to a constant. The *OBDD*-vector of the term is built using each *RegVal* of the previous cycle as *dd-cutpoint*. If each bit of the *OBDD*-vector is equivalent either to 0 or 1, then the constant result of the term is calculated;
- if the term has not changed in the last step, i.e., $RegVal\,{}_x^i \cong_C RegVal\,{}_{x-1}^i$

Otherwise equivalence to the first corresponding $RegVal_y^s$ in the specification (with the lowest y) is checked, which is neither equivalent to some term of the implementation nor to some initial *RegVal*. Consider first the case that the preceding $RegVal\,{}_{x-1}^i$ in the implementation has an equivalent "counterpart" in the specification. In this case, all *RegVals* of the preceding cycle are used as *dd-cutpoints* in the implemenation during the *dd-check*. But equivalent terms might be reached in the specification and in the implementation after different numbers of cycles.

Example 6.2
$\texttt{x}_1^s$ ←`a+b+c` *in the specification is calculated in two cycles by* $\texttt{x}_1^i$ ←`a+b` *and* $\texttt{x}_2^i$ ←$\texttt{x}_1^i$`+c` *in the implementation. Only* $\texttt{x}_2^i$ *has an equivalent counterpart in this case. The dd-check cannot reveal this fact if the dd-cutpoints are set to the previous cycle, i.e.,* $\texttt{x}_1^i$.

Therefore, a failed *dd-check* is repeated with the *dd-cutpoints* shifted succesively to the preceding cycle until either the *dd-check* is satisfied or the *RegVal* of the relevant register in the implementation has an equivalent counterpart in the specification. Note that equivalence would be revealed in the simple Example 6.2 used for illustration without *dd-check* by the other equivalence detection techniques described in chapter 5.

6.5 Considering Previous Decisions

A case-split is performed each time the value of a condition is not determined by the acceptable initial values of the registers. The decision is reflected in the *EqvClasses* and is, therefore, considered by the equivalence detection techniques during the symbolic simulation as well as during formula construction in a *dd-check*. There remain cases where the decisions have to be considered separately.

Example 6.3
The equivalence of the final values of `res` *in Fig. 6.2 is not detected without dd-check since none of the bits of the bit-vector* `m` *is constant. The dd-check has to consider the inequivalences of* `m` *and the four constants to reveal that the least significant bit of* `res` *is equivalent to 0 (see box in Fig. 6.2).*

Specification	**Implementation**
if `m=0110 or m=0011 then ...`	$\texttt{res}_1^i \leftarrow \texttt{b}[31:1]\&$`((not m[3]) and m[1]);`
elsif `m=0010 or m=0111 then ...`	
else $\texttt{res}_1^s \leftarrow \texttt{b}[31:1]\&0$;	

$$\texttt{(m}\not\cong_C\texttt{0110) and (m}\not\cong_C\texttt{0011) and (m}\not\cong_C\texttt{0010) and (m}\not\cong_C\texttt{0111)} \Rightarrow \texttt{((not m[3]) and m[1])}\cong_C 0$$

Note that none of the bits of `m` *is constant.* `m` *is declared* `m[3:0]`

Fig. 6.2: Considering decisions in a *dd-check*

Therefore, every *dd-check* which failed to demonstrate a formula $\mathcal{F}$ is repeated considering decisions about conditions which share terms with the formula. Conditions from *CondBits* which have no terms in common with the formula can have no impact on the check. The conditions of the relevant *CondBits* are combined by conjunction. Conditions of *CondBits* which are decided to be false are considered negated, see Equation 6.1.

$$dec^{rel} = \bigwedge_{i \in relevantCondBits} \begin{cases} cond\_of(i) & : \text{ if } value(i) = true \\ not\,(cond\_of(i)) & : \text{ if } value(i) = false \end{cases} \tag{6.1}$$

Note that only previous decisions are considered for intermediate *dd-checks* described in section 6.4. The repeated *dd-check* tests whether $dec^{rel} \Rightarrow \mathcal{F}$ holds.

If it is only checked whether a term is equivalent to a constant, the test has to be refined. No $\mathcal{F}$ is provided since the constant is unknown. But each bit has to be equivalent to a constant. Therefore, it is checked if dec^{rel} implies that each bit of the term is either 0 or 1:

$$\forall k \in bits\ of\ term : \left[dec^{rel} \Rightarrow dd\_of(k)\right]\ or\ \left[dec^{rel} \Rightarrow not\,(dd\_of(k))\right] \tag{6.2}$$

If there exists an equivalent constant, then it is calculated during the check of Equation 6.2. Note that accessing $dd\_of(k)$ is for free using vectors of *OBDDs*.

When building the formula for the *dd-check*, terms are often represented by *dd-cutpoints* or by other (simpler) terms in the same *EqvClass*. For example, if a term is in an *EqvClass* with a constant, then only the *OBDD* for the constant is constructed. However, these replacements have to be considered when including previous decisions.

Example 6.4
Fig. 6.3 (a) extends the example of Fig. 6.1 by an assignment $\texttt{ir}_1^s \leftarrow$`ak[1:0]` *and a condition testing* $\texttt{ir}_1^s =$*"10".*

(a)

Specification

```
ir1s ← ak[1:0];
if ir1s="10" then
  if ak[3:0]=mi[3:0]
    then ...
    else selected branch;
```

Implementation

```
cnt1i ←bit_n &...&
      ((ak[3] xor mi[3]) nor (ak[2] xor mi[2])) and
      ((ak[1] xor mi[1]) nor (ak[0] xor mi[0]))
      &...& bit_0;
```

(b) Formula used for checking if $\texttt{cnt}_1^i$ is equivalent to 0

```
ak[3:0] ≇C mi[3:0] ⇒
    ((ak[3] xor mi[3]) nor (ak[2] xor mi[2])) and          ;;not valid
    ((1 xor mi[1]) nor (0 xor mi[0]))≅C0
```

Note: $\texttt{ir}_1^s$ does not appear in the formula

(c) Refined $dec^{rel}_{refined}$ (see below) permit to obtain correct result

```
(ak[3:0] ≇C mi[3:0]) and (ak[0]≅C0) and (ak[1]≅C1) ⇒
    ((ak[3] xor mi[3]) nor (ak[2] xor mi[2])) and          ;;valid
    ((1 xor mi[1]) nor (0 xor mi[0]))≅C0
```

Fig. 6.3: Refining the decisions considered in a *dd-check*

The constants 1 and 0 are simpler to represent than the equivalent terms `ak[1]` *and* `ak[0]`*. Therefore, the constants replace those terms in the dd-check (see Fig. 6.3 (b)). But the formula in Fig. 6.3 (b) is not valid. Note that* $\texttt{ir}_1^s$ *does not appear in the formula even before replacing* `ak[1]` *and* `ak[0]` *by 1 and 0.*

The calculation of dec^{rel} has to consider that

- only one representative may be used for terms of the same equivalence class, i.e., the same vector of *OBDDs* is used and that
- decisions about the equivalence of a term (e.g., $\texttt{ir}_1^s$ in Fig. 6.3 (a)) can establish equivalences of *some* bits of another term (`ak`). Therefore, dec^{rel} is also refined if a term in a *bit-selection* is represented in the formula of the *dd-check* by another term or by a *dd-cutpoint*.

The calculation of $dec^{rel}_{refined}$ is given in equation 6.3.

$$dec^{rel}_{refined} = dec^{rel} \wedge \left(\bigwedge_{term_i \in \mathcal{R}} term_i \cong_{\mathcal{C}} repr(EqvClass(term_i)) \right) \tag{6.3}$$

The set $\mathcal{R}$ contains all terms or *bit-selections* of terms which satisfy:

- the term has been replaced by a term of the same *EqvClass* or by a *dd-cutpoint* during formula construction, i.e., it is represented in the *dd-check* by $repr(EqvClass(term_i))$, and
- one of the terms in the conditions of the relevant *CondBits* is either equivalent to the term or - if the term is a *bit-selection* - to the argument of the *bit-selection*.

Fig. 6.3 (c) describes how the correct result is obtained using $dec^{rel}_{refined}$.

6.6 Reusing Results of a *dd-check*

The result of a *dd-check* is also used in the following symbolic simulation of the remaining paths. It is likely that also in other paths a *dd-check* is invoked again to verify the same formula, which should be avoided. The corresponding decision diagram will be *not* built again since formulas are hashed as described in section 6.2. However, detecting the equivalence of the two terms already during simulation is more efficient since this information can be used to detect other equivalences and to avoid false paths.

The *EqvClasses* of the two terms tested for equivalence in the *dd-check* cannot be unified directly: the *dd-check* verified their equivalence only concerning the set of possible initial *RegVals* on a given path. Decisions resulting from case-splits might be considered in the *dd-check*. Furthermore, terms can be replaced by *dd-cutpoints* or by other equivalent terms for the construction of the decision diagrams. The following conditions have to be satisfied to reuse the result of a *dd-check*:

- the values of the *CondBits* considered in the *dd-check* must be the same,
- all terms which are replaced by the same *dd-cutpoint* in the *dd-check* are in the same *EqvClass*, and
- if a term is replaced by another equivalent term in the *dd-check* then those two terms must be once more in the same *EqvClass*.

These conditions and the equivalence they imply are the result of the *dd-check*. Verifying if the conditions hold is fast during the following simulation of the remaining paths, since only values of *CondBits* are reviewed and *EqvClasses* are compared. The conditions are checked whenever

- one of the two terms compared in the *dd-check* is found,
- one of the terms (except constants) which must be equivalent to other terms is found or its *EqvClass* is unified with another *EqvClass*, or

- one of the *CondBits* considered is decided.

The *EqvClasses* of the terms compared in the *dd-check* are unified if the corresponding conditions are satisfied.

Chapter 7

Experimental Results

Symbolic simulation has been applied to demonstrate computational equivalence of descriptions at different levels of abstraction:

- *behavioral-rtl against behavioral-rtl*: automatically constructed pipelined processors are compared to the corresponding specifications of the DLX-, Alpha,- and PIC-instruction set. The results in section 7.1 demonstrate that our symbolic simulator copes with distinct orders of memory operations in the two descriptions to be compared;

- *behavioral-rtl against structural-rtl*: the verification of a structural description of a microcontroller against two behavioral specifications is presented in section 7.2; furthermore, experimental results for the verification of pipelined DLX-processors with different implementation details are reported;

- *rt-level against gate-level*: descriptions at gate-level synthesized by the Synopsys® Design Compiler™ using the Alcatel™ MTC45000-library are compared to specifications at behavioral rt-level.

Note that most of the experiments required a *sequential* verification, e.g., equivalence of the descriptions at gate- and rt-level is only given after several control steps.

The scope of the verification tool described in section 2.2 is larger than equivalence checking. In particular, property verification is another possible application area, see section 2.7. First results about an application to another verification problem than equivalence checking are given in section 7.4 which describes register binding verification.

7.1 Behavioral RTL against Behavioral RTL

The results of the verification of four designs are given in Tab. 7.1. In all examples, a sequential specification is compared with the corresponding pipelined implementation. The specifications reflect a subset of the instructions of the respective architecture, i.e., the Alpha-architecture from Digital [Cor92], the DLX-architecture [HP96], and the PIC16C5X-processor from Microchip [Inc93]. The implementations were generated automatically from the specifications using a transformation tool developed at Darmstadt University of Technology [Hin00, HER99, HRE99, EHR98, Hin98a, HRE00]. Note that there are considerable differences between our Alpha implementation and the processor produced by Digital, e.g., concerning the number of pipeline stages. The TUD transformation tool uses a small-set of correctness preserving transformations. The descriptions are obtained automatically by gradual application of the transformations. Another application of the TUD transformation tool in addition to pipeline synthesis is the automatic verification of scheduling results in high-level synthesis [EHR99].

Verification of the pipelined designs was done using the flushing approach of [BD94], see section 4.1.3 and appendix 9.8. The two acyclic finite sequences to be compared are generated automatically. No transition function is required as in [BD94]. The behavioral description of the pipelined system consists of several parts, called segments, which describe different combinations of instruction stages of the pipeline, i.e., the partially filled/flushed pipeline or the full pipeline state. All parts have to be verified using the flushing approach. For example, 9 parts are used to describe a DLX with 5 pipeline stages [Hin00]. The behavior of the system if the stall-input is set or cleared, i.e., whether the pipeline is flushed or not is separately described for each part. Therefore, *automatic* generation of the two sequences to be compared (section 4.1.3) is possible by setting the stall-input accordingly and simply linking the relevant parts until the part describing the empty pipeline is reached. An example is given in [Hin00].

Tab. 7.1 gives the verification time, the number of instruction classes[1], and the total number of paths checked during the symbolic simulation of all parts of the descriptions.[2] Computational equivalence has been verified with respect to the data memory, the register file, and the program counter.[3] Measurements are on a Sun Ultra II with 300 MHz.

The results demonstrate that the equivalence detection techniques described in section 5.9 cope with distinct orders of memory operations in the two descriptions to be compared. The sixth column shows in how many paths `store`-operations

[1] For example, the instruction classes of the DLX are direct and indirect `alu-`, `load-`, `store-`, `branch-`, and `jump`-instructions.

[2] The verification results for the different parts are aggregated in Tab. 7.1. [Hin00] reports the results for each part separately.

[3] The instruction memory is not written, i.e., verification is trivial.

are overwritten (section 5.9.3, pages 90 to 91). The number of paths with changed order of the `store`-operations (section 5.9.3, pages 91 to 94) is given in the last column. Paths with changed `store`-order are not considered in the sixth column although `stores` may be overwritten in these paths, too.

The `store`-order in the DLX-example is always identical in the specification and in the implementation and no overwritten `stores` have to be considered. The same results have been obtained for the verification of the structural DLX-descriptions, see section 7.2. The Alpha-example requires additionally detecting overwritten `store`-operations. Consider two `stores` to the Alpha register file with equivalent addresses, which are executed consecutively in the sequential description. One of them is skipped if they are executed in different instruction stages which are parallelized by the synthesis tool. Note that the register file of the DLX (respectively the data memory) is always written in the same instruction stage.

Description	Pipeline stages	Instruction classes	Verification time	Total paths	Paths with `stores`	
					overwritten	changed order
DLX	5	6	46min 23s	1506698	-	-
Alpha	3	10	7.84 s	2374	88	-
PIC 1	2	17	252.6 s	107655	3151	1741
PIC 2	2	17	379.6 s	161622	4338	5252

Tab. 7.1: Experimental results for behavioral rtl verification

All techniques presented in section 5.9 are required to verify the two PIC-examples. The `store`-order was changed significantly in many paths after introducing pipelining. The reason is the data memory mapping used by this architecture, i.e., single registers are addressed in the same manner as registers of the register file. Formal verification has to consider the access to registers and register file by a single memory model, see also [Hin00] and section 5.9. The mapping makes synthesis (and verification) more complicated since numerous additional data conflicts have to be resolved. This is also demonstrated by the higher complexity of PIC 2 compared to PIC 1. The only difference of PIC 1 is that the STATUS-register is excluded from data mapping. Another reason for the complexity of the PIC-examples compared to the Alpha- and DLX-example (which have more pipeline stages) is the larger number of instruction classes. The DLX-results reported in [RHE99] refer to a simpler DLX-description[4] than the results given in Tab. 7.1.

We verified the Alpha-example with the test for changed `store`-order switched off and the DLX-example also without the checks for overwritten `stores`. Computation time changed only less than one second, which demonstrates that the overhead introduced by testing for complex `read`/`store`-schemes in the equivalence detection is acceptable.

[4]The instruction stages are less parallelized and the description considers one instruction class less.

Only the verification of the PIC-examples required *dd-checks* as described in chapter 6 to demonstrate computational equivalence. The small set of transformations used during synthesis is considered by the other equivalence detection techniques. The *dd-checks* in the PIC-example are only necessary to detect inconsistencies due to the varying length of the PIC instruction code.

Various *implementation* bugs of the synthesis tool have been revealed by the symbolic simulator. These bugs did not concern the correctness of the transformations but only the implementation of the tool. One of the bugs which has been detected during the verification of the behavioral DLX-processor is illustrated in the following example.

Example 7.1
The abbreviations in Fig. 7.1 denote the instruction stages of the DLX-pipeline, i.e., instruction-fetch- (IF), decode- (ID), execute- (EX), memory- (MEM) and write-back-stage (WB). The error occurs iff an ALU-instruction uses both values loaded by two directly preceding LOAD-instructions. The execution on the initially generated system with pipelining is described in Fig. 7.1. The ALU-instruction is stalled until the preceding LOAD-instruction has reached the WB-stage. The first LOAD-instruction terminates in the meantime. The ALU-instruction has loaded in its ID-stage the old, wrong value of `R1`*. But it is not possible to forward the correct value in the EX-stage since the first LOAD-instruction has terminated by writing the value of* `R1` *into the register-file. The synthesis tool did not detect the data dependency in an older version. Solutions to avoid this bug are to repeat the ID-stage during the stall or to add another pipeline register.*

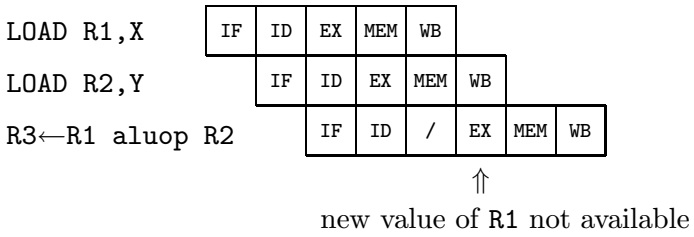


Fig. 7.1: Implementation bug revealed

A practical important advantage of the symbolic simulator is its good debugging support.

Example 7.2
The following comments about a counterexample turned out to be helpful during the experiments reported in this section:

- *a complete description of the path in specification/implementation in the initial and/or the internal description language;*

- *the last expressions assigned to the registers and memories with and/or without backward-substitution of expressions assigned to RegVals;*
- *decisions performed at case-splits, i.e., the values of decided CondBits; additionally, a list of the values of all CondBits, i.e., including conditions not requiring a case-split;*
- *equivalence of RegVals/terms to constants;*
- *a summary how instructions are carried through the pipeline registers;*
- *inequivalences of EqvClasses.*

The symbolic simulator provided also useful information for the improvement of the synthesis tool after a *successful* verification. Never taken branches of *if-then-else*-clauses are reported after simulating symbolically all possible paths. These branches are logically impossible and indicate redundancy of the control logic.

7.2 Structural RTL against Behavioral RTL

7.2.1 DLX-Processor Descriptions

Two implementations of a subset of the DLX processor [HP96] have been verified, the first from [HSG98], initially verified in [BD94], and a second one designed at Darmstadt University of Technology.[5] The second description contains more structural elements, e.g., multiplexers and corresponding control lines required for forwarding are given. Both examples have a 5-stage pipeline with *branch-predict-not-taken* strategy.[6]

For both descriptions, acyclic sequences are generated by using the flushing approach of [BD94]; i.e., the execution of the inner body of the pipeline loop followed by the flushing of the pipeline is compared to the flushing of the pipeline followed by one serial execution. Different from [BD94] (see also [Bur96]), our flushing technique guarantees that one instruction is fetched *and* executed in the first sequence. Otherwise it has to be communicated between the specification and the implementation if an instruction has to be executed on the sequential processor or not (e.g., due to a load interlock in the implementation). [Bur96] describes this as keeping the implementation and the specification in sync. How to generate the two finite sequences to be compared using the flushing approach of [BD94] is described for the second structural DLX-example in appendix 9.5.

[5] A slightly modified version of the second design has been verified, too, which is not discussed in the following. The only difference of the modified design is that branches are taken in the EX-stage instead of the ID-stage.

[6] The separation of writing to and reading from the register file is modeled in *LLS* by an additional segment for the register writing.

Verification is done automatically, only the (simple) correct flushing schema, guaranteeing that one instruction is fetched and executed, has to be provided by the user. In addition, some paths are collapsed by a simple annotation that can be used also for other examples. Forwarding the arguments to the ALU is obviously redundant, if the EX-stage contains a bubble (NO_OP) or a branch. `Unknown`-terms are used in these cases, i.e., the value of the ALU-inputs is set to a distinct unknown value, see section 5.8. The verification remains *complete*, because the *EqvClasses* of the final *RegVals* to check would always be different, if these final *RegVals* depend on one of the distinct `unknown`-terms. Note that verification has been done for both examples also without this annotation, but with $\approx 90\%$ more paths to check.

Version	paths	aver. time per path	total time
DLX from [HSG98]	310,312	12.6 ms	1h 5min 13s
DLX with multiplexers	259,221	19.5 ms	1h 24min 14s

Tab. 7.2: Experimental results for structural DLX verification

Two errors introduced by the conversion of the data format used by [HSG98] and several bugs in our hand crafted design have been detected automatically by the symbolic simulator. Verification results of the correct designs are given in Tab. 7.2. Measurements are on a Sun Ultra II with 300 MHz. Note that the more detailed and structural description of the second design does not blow up verification time: increase of the average time per path is acceptable. The number of paths remains nearly the same (even decreases slightly due to a minor different realization of the WB-stage).

Verifying the DLX-examples does not require *dd-checks*. The pipelined implementations can be derived from the sequential specifications with exception of the multiplexers in the second design mostly by scheduling and without, e.g., considering bit-vector arithmetic operations, see also the DLX-example in section 7.1. Verifying examples like the DLX is not the main intention of our approach since the capabilities of the symbolic simulator are only partly used. But they demonstrate that also control logic with complex branching can be verified by symbolic simulation.

7.2.2 Microprogram-Control with and without Cycle Equivalence

In this example, two behavioral descriptions of a simple architecture with microprogram control are compared to a structural implementation. The microprogram control is performed in both behavioral descriptions by simple assignments and no information about the control of the datapath-operations, e.g., multiplexer-control is given. The structural description of the machine contains

an ALU, 7 registers, a RAM, and a microprogram ROM. All multiplexers and control lines required are included. The two behavioral descriptions differ in the number of cycles for execution of one instruction:

- the first is cycle equivalent to the structural description; i.e., the values of the registers are equivalent in every step. The description consists of a "big" *if-then-else*-clause where every branch considers the microprogram-step for a distinct value of the microprogram counter. The finite sequences to be compared are simply the respective loop-bodies describing one microprogram step;

- the second behavioral description is less complex than the first and more intuitive for the designer. It contains an instruction fork in the decode phase. No cycle equivalence is given. Therefore, the sequences to be compared are the complete executions of one instruction, i.e., a sequential verification is necessary. The only annotation of the user concerns the constant value of the microprogram counter in the structural implementation, that indicates the completion of one instruction. Furthermore, the number of cycles to simulate has to be provided. Appendix 9.5 describes the finite sequences to be compared and the annotations in more detail.

The ROM is represented by one multiplexer with constant inputs. In this example, the read/write-schema used also by SVC would not work, since the ROM has constant values on all memory-places. The ROM accesses and the other multiplexers would lead to term-size explosion if they are interpreted as functions (canonizing!) by formula based techniques, see section 3.3. The same holds if they are considered as *if-then-else*-clauses, since symbolic simulation goes over several cycles in this example.

Example	paths*	*dd-checks*	false paths	time
with cycle equivalence	291	56	39	24.53s
different number of cycles	123	41	16	19.58s

* including false paths

Tab. 7.3: Experimental results for microprogram-controller verification

Results are given in Tab. 7.3. Measurements are on a Sun Ultra II with 300 MHz, verification times include the construction of decision diagrams. The third column indicates how often the *dd-checks* of chapter 6 are used either to demonstrate equivalence or to detect an inconsistent decision, i.e., one of the false paths reported in the fourth column is reached. Mainly bit-selections from the ALU-output caused *dd-checks*, i.e., application of bit-vector arithmetic has to be revealed. The in principle more difficult verification without cycle equivalence requires less paths since the decisions in the behavioral description determines the path in the structural description.

Verifying the designs requires an unnecessarily great number of paths, if the value of intermediate carriers[7] or registers representing single bit control signals is not decided. These control signals appear frequently in complex conditions. Often the value of those conditions cannot be determined if the control signals are not equivalent to either 0 or 1. The following case-split leads frequently to an inconsistent decision which has to be revealed by a *dd-check*. Therefore, a decision about the value of the single bit control signals is forced instead of case-splitting at the complex conditions. This is achieved by transforming automatically during pre-processing, for example, `ctrl←a or b;` to `ctrl←`**if** `a or b` **then** `1` **else** `0`. Again, no insight into the automatic verification process is required.

7.3 Gate-level against RT-level

Two types of examples have been examined, a simple read/write-architecture (RWA), which takes three cycles to execute an instruction and a more complex architecture with microprogram control (MPA). Two specifications of the second architecture without cycle equivalence are given; only the first is used for synthesis; therefore, it is cycle equivalent to the synthesis result. Verification of the gate-level implementation against the other specification without cycle equivalence requires a sequential verification since the complete execution of an instruction has to be compared.

The gate-level descriptions of both examples are generated using the Synopsys® Design Compiler™ with the Alcatel™ MTC45000-library. All memory operations are replaced by assignments to interfaces before synthesis, see appendix 9.4. Equivalence of memory operations on these ports has been verified according to [RHE99], too. The automatic compilation of the synthesis results into our internal description language is described in appendix 9.4. All transformation steps are summarized for the MPA example in appendix 9.9.

The MPA synthesis result comprises 927 standard cells, two arithmetic units, and one incrementer. The standard cells except the arithmetic blocks and the memory are broken internally into basic Boolean functions with up to 4 inputs, see section 5.2 and appendix 9.4.

Tab. 7.4 summarizes the results. All our measurements are on a Sun Ultra II with 300 MHz. Four equivalence checks have been performed:

(1) one cycle RWA^{RTL} against one cycle RWA^{gate};

(2) one instruction (3 cycles) RWA^{RTL} against one instruction RWA^{gate} (with also 3 cycles);

(3) one cycle synthesizable specification $\mathrm{MPA}^{RTL}_{cycle}$ against one cycle MPA^{gate};

[7] They are represented as *simulation-cutpoints* and considered during simulation as "artificial" *RegVals*, see appendix 9.3.

(4) one instruction with $m \leq 8$ cycles in the non-synthesizable specification without cycle equivalence $\text{MPA}^{RTL}_{non-cycle}$ against one instruction in MPA^{gate} with $n \leq 10$ cycles ; m and n depend on the instruction and may be different.

check number	cycles spec	cycles impl	Verification time	*dd-checks*
(1) RWA (one cycle)	1	1	1.7s	-
(2) RWA (one instruction)	3	3	5.5s	-
(3) MPA (with cycle-equiv.)	1	1	74 s	13
(4) MPA (w/o cycle-equiv.)	≤8	≤10	786 s	92

Tab. 7.4: Experimental results for rtl⇔gate-level verification

The verification time given in Tab. 7.4 increases for both designs acceptably with the number of sequential steps simulated. Especially the last check would lead to term-size explosion if a formula is built in advance and evaluated afterwards, since the whole gate-level expressions of a cycle represent the arguments in the next cycle. The number of *dd-checks* performed during symbolic simulation is given in the fifth column of Tab. 7.4.

Example 7.3
The following equivalences had to be revealed by dd-checks during the verification of the MPA-example. 0/1 stand for complex terms which have been detected in this path previously to be equivalent to 0/1:

- *absorption, e.g.,*

  bit_{31} `& (not (1 nand (((AK[30] and 1) or 0) nand MI[30]))) nand (0 nor ((AK[30] and 1) or 0)) & ... & ` bit_0
  $\cong_{\mathcal{C}}$ `AK`

- *Boolean datapath-operations on bit-vectors, e.g.,*

  bit_{31} `& 1 nand ((1 and (MI[30] xor AK[30])) nor 0) & ... & ` bit_0
  $\cong_{\mathcal{C}}$ `((vnot AK) vand MI) vor (AK vand (vnot MI))`

  where `vand` *etc. are Boolean operations on bit-vectors;*

- *the examples in Fig. 6.1, 6.2, and 6.3.*

All extensions of the *dd-checks* described in section 6.4 are used, which are not necessary for the experiments reported in the previous sections.

Example (4) was also checked using *only* vectors of *OBDDs* at the end of a path. The information of the other equivalence detection techniques of chapter 5 was *not* evaluated in contrast to the experiments reported in Tab. 7.4. Verification ran out of memory.

Verification was automatic, the only user-annotations concern the completion of an instruction for check (2) and (4) and the designation of the 3 (RWA) respectively 5 (MPA) control registers for intermediate *dd-checks* (section 6.4).

7.4 Example of Further Applications: Register Binding Verification

Register binding verification is an example of the application of the symbolic simulator to another verification problem than equivalence checking. The approach presented first in [Bla00, BRHE00] combines symbolic simulation and model checking. A brief overview is given in the following.

Register binding determines how several variables of a design can share a common register to minimize costs. A register binding is correct if no conflicts of variables mapped on the same register exist. A conflict occurs if the value of a variable is overwritten before it was referenced the last time. Binding algorithms utilize that conflicts on logically impossible paths are irrelevant.

Conflicts can be expressed as CTL formulas [EC80] which are checked by means of symbolic model checking [BCL$^+$94]. As all techniques which depend on state space exploration, symbolic model checking faces the problem that the number of states grows generally exponentially with the number of storage elements, see section 3.5. A solution to the state explosion problem for register binding verification is to abstract all data operations, particularly bit-vector operations. Counterexamples given by the model checker may be false negatives due to this abstraction, e.g., if the control flow depends on arithmetic bit-vector operations. Therefore, a *reduced* description with the marked conflict paths is generated and symbolically simulated. The symbolic simulator uses no abstraction and can determine by checking a monitor-register if one of the conflict paths is possible, i.e., if the register binding is in fact not correct.

Example 7.4
The variables `RADDR1` *and* `RADDR2` *in Fig. 7.2 (a) are mapped onto the same register* `REG`*. Both variables can be assigned (GEN) in segment* `L1` *and used in the subsequent segment* `L2` *(USE).*

A conflict would occur, if one of the branches in `L1`*, where* `RADDR1` *is assigned, is reached and then the* `then`*-branch of* `L2` *is taken, where* `RADDR2` *is used (and vice versa). But all conflict paths are logically impossible. For example, if the first branch in* `L1` *is taken, then* `P[0:1]`$\cong_C$`00` *holds and* `P[0]`$\cong_C$`0` *is assigned to* `Z`*. The* `then`*-branch of* `L2` *with the conflict cannot be reached since* `Z` *is equivalent to 0. The register binding is correct since all other conflicts are on logically impossible paths, too. The model checker cannot identify the contradictions due to the data abstraction. Therefore, the description in Fig. 7.2 (b) is generated to verify the conflicts by symbolic simulation. Note that the description to simulate is not reduced* in this example *since conflicts are detected by the model checker in all branches.*

Two monitor registers `REG` *and* `CHECK` *are added in Fig. 7.2 (b).* `REG` *is set to the same value as* `RADDR1` *or* `RADDR2` *whenever they are assigned. Each time one of the variables* `RADDR1` *and* `RADDR2` *is used, it is tested if the value of the*

(a) *LLS* description

```
L0: (IR←INC(DATA));
    (P[0:1]←IR[0:1]);
    L1;
L1: IF P[0:1]=00 THEN
        (ADDR←ADD(STACK,OFF1));
        (RADDR1←STACK);      – GEN RADDR1
        (Z←P[0]);
    ELSIF P[0:1]=10 THEN
        (ADDR←INC(STACK));
        (RADDR1←STACK);      – GEN RADDR1
        (Z←P[1]);
    ELSIF P[0:1]=01 THEN
        (ADDR←OFF1);
        (RADDR2←ADDR);       – GEN RADDR2
        (Z←P[1]);
    ELSE
        (ADDR←00000000);
        (RADDR2←ADDR);       – GEN RADDR2
        (Z←P[0]);
    ENDIF;
    L2;
L2: IF Z THEN
        (MADDR←ADDR);
        (RADDR←RADDR2);      – USE RADDR2

    ELSE
        (MADDR←ADD(ADDR,OFF2));
        (RADDR←RADDR1);      – USE RADDR1
    ENDIF;
    L0;
```

(b) Implementation for symbolic simulation

```
(CHECK←0);
(IR←INC(DATA));
(P[0:1]←IR[0:1]);

IF P[0:1]=00 THEN
    (ADDR←ADD(STACK,OFF1));
    (RADDR1←STACK, REG←STACK);
    (Z←P[0]);
ELSIF P[0:1]=10 THEN
    (ADDR←INC(STACK));
    (RADDR1←STACK, REG←STACK);
    (Z←P[1]);
ELSIF P[0:1]=01 THEN
    (ADDR←OFF1);
    (RADDR2←ADDR, REG←ADDR);
    (Z←P[1]);
ELSE
    (ADDR←00000000);
    (RADDR2←ADDR, REG←ADDR);
    (Z←P[0]);
ENDIF;

IF Z THEN
    (MADDR←ADDR);
    (RADDR←RADDR2,
     CHECK←CHECK or VIOLATE(RADDR2,REG));
ELSE
    (MADDR←ADD(ADDR,OFF2));
    (RADDR←RADDR1,
     CHECK←CHECK or VIOLATE(RADDR1,REG));
ENDIF;
```

(c) Specification for symbolic simulation

```
(CHECK←0);
```

RADDR1 and RADDR2 are supposed to be mapped onto the same register. Figure taken with slight modifications from [Bla00]. The example is taken from [ABRM98] and [Ber91].

Fig. 7.2: Example for register binding verification

variable and REG *are equivalent.* VIOLATE *supplies 0 iff both RegVals are in the same EqvClass. A conflict is possible if a path is simulated where* CHECK *is set to 1 at least once.*[8] *This is tested by checking computational equivalence to the specification in Fig. 7.2 (c) containing only an initialization of* CHECK *to 0; i.e., the verification problem is reduced to an equivalence check, see also section 2.7. Note that the special case is considered, where the same value is assigned to both variables and, therefore, a conflict is irrelevant.*[9]

No false negatives are produced. The technique is currently limited by conflicts

[8] The disjunction prevents resetting CHECK.

[9] For example, RADDR1 is assigned to REG and then RADDR2 is used. Mapping onto the same register does not lead to an erroneous behavior, if the values of RADDR1 and RADDR2 are not distinguishable, i.e., the *RegVals* are equivalent.

encountered in loops. Correctness is guaranteed in these cases only for the sequential depth of the symbolic simulation. However, if no conflicts *in loops* are detected *by the model checker* then the binding is guaranteed to be correct for an arbitrary sequential depth, see [Bla00, BRHE00].

Applying model checking previously instead of using only symbolic simulation has two advantages. The descriptions to be simulated symbolically are reduced; i.e., branches where model checking reasoning on the simpler abstraction model does not encounter conflicts need not be taken into consideration. Furthermore, symbolic simulation reasons about a finite number of steps while model checking can consider, e.g., an arbitrary number of loop iterations.

Verification is performed automatically, and is independent of the applied register binding technique.

Chapter 8

Conclusion

A new approach for the automatic formal verification of digital systems by symbolic simulation is presented. Experimental results demonstrate the applicability to *sequential* equivalence checking at different levels of abstraction although our examples are still not nearly as complex as commercial designs. The equivalence of structural descriptions at rt-level with implementation details and their corresponding behavioral specifications is demonstrated. Gate-level results of a commercial synthesis tool are compared to specifications at behavioral or structural rt-level. The specification need not be synthesizable nor cycle equivalent to the implementation. The symbolic simulator supports a different number of control steps in the two descriptions to be compared. Automatic equivalence checking is independent of the specific synthesis tool and copes also with manual modifications by the designer.

Symbolic values are used for registers and memories instead of test-vectors to permit a complete verification. Simulation is guided along valid, i.e., logically consistent paths in the descriptions. Indeterminate branches, that depend on initial register or memory values, are considered by case splits to check for an arbitrary control flow. Several register assignments along a valid path are explicitly distinguished instead of rewriting the register with the expressions assigned to it. Therefore, term-size explosion is avoided.

In contrast to previous approaches, symbolic terms are never modified during simulation, e.g., by canonizing or rewriting them. No unique representation is required. Instead, the results of the equivalence detection techniques are marked at equivalence classes. This permits a flexible use of an open library of different equivalence detection techniques in order to find a good compromise between accuracy and speed. New techniques can easily be added to this hierarchical equivalence detection organized according to the principle of Hennessy and Patterson [HP96]: "*Make the common case fast*".

An effective combination of symbolic simulation and decision diagrams was implemented which permits detecting corner-cases of equivalence. Only small parts of the verification problem are reflected by decision diagrams since the

results of the other equivalence detection techniques are used. Therefore, graph explosion is avoided and a *sequential* verification of a design at gate-level against a specification at rt-level is possible. Furthermore, functions that are worst-case exponential with *OBDDs*, e.g., multiplication can be left uninterpreted during the decision diagram based checks.

Symbolic simulation has to cope with memories of arbitrary size. Modeling memory access by array operations solves the size problem, but makes the detection of equivalent array operations necessary in order to capture the functionality of the memory. A reasoning process about the relationships of addresses is required, since they can be arbitrary symbolic terms. Collecting equivalent symbolic terms in equivalence classes permits to establish a fast address comparison for our equivalence detection method. The new technique makes possible an efficient automatic equivalence checking of descriptions with complex reorderings of memory operations.

A future application of the symbolic simulation approach to property verification is proposed. First results are given for the example of register binding verification.

An important advantage of the tool is the good debugging support. Meaningful information about a counterexample can be provided by a technique which is intuitive to the designer: simulation *"is a natural way engineers think"*.

Chapter 9

Appendix

Appendix 9.1 to 9.3 present additional transformations performed by the *FDS-to-EDS* compiler during pre-processing which are not described in section 4.1.4 to 4.1.5. Appendix 9.4 gives a brief overview of the translator which permits to verify synthesis results from the Synopsys® Design Compiler™ in VHDL-format.

Appendix 9.5 discusses two examples (DLX and microprogram architecture) for annotations of descriptions in *LLS* to generate the acyclic finite sequences for symbolic simulation as described in section 4.1.3.

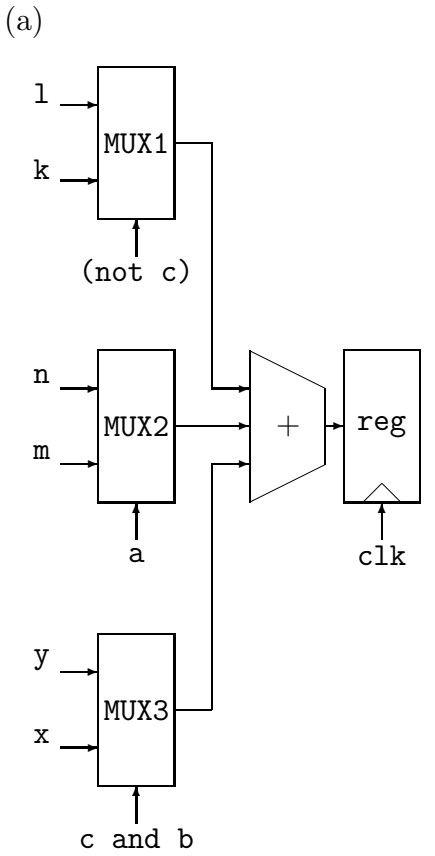
Section 9.6 summarizes the functions supported by the symbolic simulator. The tables in appendix 9.7 describe the properties of *EqvClasses*, *CondBits*, *Term Representatives*, and *RegVals*. Appendix 9.8 summarizes the approach of [BD94] for verification of systems with pipelining, see also section 4.1.3. The transformation steps for the verification of the MPA example in section 7.3 are illustrated in appendix 9.9. Finally, appendix 9.10 lists some implementation details which have been tested and rejected, or which have been improved during the development of the symbolic simulator.

9.1 Extracting ITE-Clauses in Functions

Arguments of functions can contain *if-then-else*-clauses in *LLS*. Fig. 9.1 (b) gives an example for the behavioral description of the multiplexer/adder-combination shown in Fig. 9.1 (a).

If-then-else-clauses describe mostly the control part of a description. If their condition cannot be decided but depends on the initial *RegVals* then a case-split should be performed during symbolic simulation. Otherwise equivalence detection fails too often since no equivalent terms exist mostly if the arguments contain symbolic *if-then-else*-clauses with conditions that are not decided.

Performing the case-split during symbolic simulation *while tracing the arguments of functions* is not efficient with regard to the simulation speed. A backtracking of the symbolic simulation would become necessary if parallel assignments have to be considered or if a function has more than one argument.



(b)

```
reg ←add( (if (not c) then l else k),
          (if a then n else m),
          (if (c and b) then y else x) );
```

(c)

```
reg ←
   if (a and (not c)) then add(l,n,x);
   elsif (a and c and b) then add(k,n,y);
   elsif (a and c and (not b)) then add(k,n,x);
   elsif ((not a) and (not c)) then add(l,m,x);
   elsif ((not a) and c and b) then add(k,m,y);
   else add(k,m,x);
```

Fig. 9.1: Extracting *if-then-else*-structures in arguments

Furthermore, determining the point of a case-split becomes complex when saving and restoring the context. Finally, the same case-split may be required for more than one argument of a function. For example, the value `c` is used in the condition of two arguments in Fig. 9.1 (b).

Therefore, all *if-then-else*-clauses in arguments of functions are extracted during pre-processing. The conditions of the arguments are collected first and then the appropriate *if-then-else*-clause is built. Fig. 9.1 (c) shows the result. The new conditions in Fig. 9.1 (c) are conjunctions of the conditions in Fig. 9.1 (b). These conjunctions are often simplified. Impossible branches are omitted. For example, the `add`-term in Fig. 9.1 (b) contains three conditions which would lead to $2^3 = 8$ different branches. But the combinations `add(l,m,y)` and `add(l,n,y)` are not possible since `not(c)` and `c and b` cannot be satisfied both. Such mutual exclusions have to be considered already during the extraction of the conditions to avoid case-explosion. For example, if each of the three inputs of the adder would depend on which of 8 possible operation codes is valid, then this leads to $3^8 = 6561$ combinations although only 8 cases have to be distinguished. Therefore, conditions are compared already during the extraction.

Some Boolean simplifications are included in the *FDS-to-EDS* compiler. Optionally, the more powerful *Simple*-tool [HRE00] can be used which performs

false-path elimination and simplification of sequential acyclic descriptions with complex branching logic. This tool copes with sequentially dependent branching conditions involving bit-vector expressions. Note that if the *Simple*-tool has already been used to optimize the description in *IDS*-format then the built-in Boolean simplifications of the *FDS-to-EDS* compiler are generally sufficient. A repeated application of the *Simple*-tool is redundant in this case. The same holds for structural descriptions which have been simplified previously, e.g., the results of commercial synthesis tools.

9.2 Representatives for Terms

Every distinct term and subterm is replaced during pre-processing for technical reasons by an arbitrary chosen distinct variable called *Term Representative*. A new *Term Representative* is introduced for each term where the function type or at least one argument is distinct, e.g., $\texttt{pc}_1^s\texttt{+2}$ and $\texttt{pc}_2^s\texttt{+2}$ are distinguished. *Term Representatives* are introduced for each subterm.

Example 9.1
The Term Representatives `repr1` *to* `repr4` *are introduced for the term assigned to* `reg` *in Fig. 9.2. Note that bit-selections, e.g.,* `a[0:5]` *are also interpreted as functions.*

```
reg←(a[0:5]>>2)+(x<<c);
        ⇓
reg←repr1;
```

representative	associated with
repr1	repr2+repr4
repr2	(repr3>>2)
repr3	a[0:5]
repr4	(x<<c)

Fig. 9.2: Introduction of representatives for terms

The introduction of *Term Representatives* is only an implementation decision. They permit to manage the properties of a term, e.g., its *EqvClass* or if the term has already been detected on a path.

9.3 Miscellaneous Modifications

The major miscellaneous modifications are described in the following:

- *if-then-else*-clauses in conditions of other *if-then-else*-clauses are extracted, see. Fig. 9.3;
- *LLS* permits to declare *LLS-Macros* which represent an expression without register assignments. Each *LLS-Macro* in the descriptions is simply replaced by the corresponding expression;

```
if (if c1 then c2 else c3)
  then reg←x;
  else reg←y;
```

becomes

```
if c1
  then if c2
       then reg←x;
       else reg←y;
  elsif c3 then reg←x;
  else reg←y;
```

Fig. 9.3: Extracting *if-then-else*-clauses in conditions

- *simulation-cutpoints* (not to be confused with *dd-cutpoints* described in section 6.2) can be introduced if a *LLS-Macro* is used more than once to avoid multiple evaluation of the corresponding expression on the same path. The expression is assigned to the *simulation-cutpoint* before the first use of the *LLS-Macro* in the description, see Fig. 9.4. The *simulation-cutpoint* is

```
if c<5 then x1^i←0; else x1^i ←1;
...
simcut2^i:= if a1^i xor b1^i then e else f;
if (x1^i or (simcut2^i<10) or (simcut2^i>15)) then ...
```

Fig. 9.4: Example of a *simulation-cutpoint*

  interpreted in the following as an "artificial" register, which is used for the *LLS-Macro*-expression. This expression is only evaluated if the *simulation-cutpoint* is used in fact on the actual path. For example, the expression assigned to $\texttt{simcut}_2^i$ in Fig. 9.4 is not examined if $\texttt{x}_1^i$ is equivalent to 1.

  Simulation-cutpoints are introduced before indexing the *RegVals* (see section 4.1.4) since generally their expressions contain registers. Therefore, their different values have to be distinguished by indexing, too. Note that the introduction of *simulation-cutpoints* is optional. They are redundant if the expression can be represented by a single *Term Representative* since it contains no *if-then-else*-clause;

- one-bit registers or *simulation-cutpoints* (see above) *at rt-level* are often part of the control of the design. Forcing a decision about whether they are equivalent to 0 or 1 can be advantageous to detect equivalences of terms using this control register as argument, see also section 7.2.2. This is done by replacing an assignment

  `onebitreg`←*Boolean expression* by

  `onebitreg`←**if** *Boolean expression* **then** 1 **else** 0

  This transformation is optional;

- the least significant bit (LSB) has to stand on the left in the descriptions; otherwise time-consuming transformations are necessary during symbolic simulation in order to use the TUDD-package including its extension for

OBDD-vectors. Furthermore, the LSB must have the index 0. If these conditions are not satisfied then the necessary modifications concern mainly bit-selections of the register.

Example 9.2
If a register is defined initially with LSB right and an index [4:10], then the following transformations are necessary during pre-processing:

- `r[8]` *becomes* `r[2]`
- `r[5:6]` *becomes* `r[4:5]`

Note that successive *bit-selections*, e.g., `(a[3:7])[1:2]` can make these transformations complex;

- *if-then-else*-clauses in expressions assigned to registers are extracted, e.g.,

  `reg`←**if** `a` **then** `b` **else** `c` is transformed to **if** `a` **then** `reg`←`b` **else** `reg`←`c`

  This transformation is not considered in Fig. 9.1 (c) of appendix 9.1; *if-then-else*-clauses in *simulation-cutpoints* are not extracted, i.e., the assignment to $\texttt{simcut}_2^i$ in Fig. 9.4 is not modified;

- the control of a multiplexer (see section 5.4) can consist of comparing an expression to constants. The single control lines are extracted if the expression is a concatenation and the number of concatenation operations corresponds with the multiplexer size. For example, the control lines obtained from `c&b&a` for a 8:1 multiplexer are `c`, `b`, and `a`. Otherwise bit-selections are necessary to obtain the single control lines, e.g., `(a+b)[2]`, `(a+b)[1]`, and `(a+b)[0]`;

- *LLS* distinguishes whether the data inputs of multiplexers are single bits or bit-vectors, which is not required for symbolic simulation;

- Boolean functions in *LLS* have only two arguments while the number of arguments is not restricted by the symbolic simulator. Successive applications of the same Boolean function are transformed into a single application. For example, `(and (and a b) c)` becomes `(and a b c)` to reduce the number of function calls during symbolic simulation;

- all *CondBits* with a mutual exclusive condition are determined during preprocessing for each *CondBit*. If the value of a *CondBit* is set *true* during symbolic simulation then the value of all *CondBits* with a mutual exclusive condition is set *false*;

- array operations are performed in *LLS* by using the `SELSLICE2` function; they have to be transformed to `read`- and `store`-operations as described in section 4.1.5;

- constant bit-vectors are represented *internally* by integers; the length of the initial bit-vector need not be notified: a constant is either compared or assigned to a term or a *RegVal*; their length is available during symbolic simulation. Compatibility of the bit-vector length is checked during pre-processing;
- the concatenation is expressed recursively, i.e., `X & Y & Z` in VHDL is expressed as `(CAT X (CAT Y Z)` in *IDS*, see also section 5.6;
- the information about parallel or sequential execution of assignments is removed after indexing the *RegVals*, see section 4.1.4;
- some functions are expressed by other functions, e.g., a left-shift shifting in `1` is transformed into a combination of *bit-selection* and concatenation `lsh(a,1)` → `a[30:0]&1`;
- other minor syntactic transformations.

9.4 The *SYN2IDS* Translator

The *SYN2IDS* translator takes as input the standard-cell/gate-level results of the Synopsys® Design Compiler™ using the Alcatel™ MTC45000-library.[1] The output is in *IDS*-format, see section 4.1.2. Only a subset of the output format of the Synopsys® Design Compiler™ is supported.[2]

The standard cells, e.g., an `AO2`-cell are currently broken during pre-processing using basic Boolean functions, i.e., `(A and B) nor (C and D)`. Simulation speed can be optimized by providing specialized equivalence detection routines for those standard cells, too.

Specific equivalence detection techniques exist already for a subset of the (generic) arithmetic blocks of the DesignWare®-library used by the Synopsys® Design Compiler™. The synthesis output comprises the entities and architectures of the arithmetic blocks generated,[3] which are *not* translated by the *SYN2IDS* translator. A behavioral description of those arithmetic blocks is used instead. For example, an adder without carry is simply described as `(addmod a b)` without considering the structural description of the adder. Equivalence of the structural implementation of the arithmetic blocks and the behavioral description can be demonstrated, e.g., using *OBDDs*.

The single bits of a register can be recognized in the gate-level description since the first part of the names of the respective signals is identical to the register name. For example, `PC_reg_7_label` is the eighth bit of the register `PC`. Those

[1] The *SYN2IDS* translator can easily be extended to support other libraries or additional standard cells.

[2] For example, not all components of the DesignWare®-library are supported.

[3] Using the Alcatel™ MTC45000-library.

bits are concatenated in the *IDS*-format to a single term which is assigned to the respective register.

Example 9.3
Fig. 9.5 describes the transformation for the register `PC` *implemented by eight D-Flipflops. The proxies term(*`n569`*), term(*`n570`*) etc. in Fig. 9.5 represent the corresponding Boolean terms or outputs of standard cells assigned to the signals* `n569`, `n570` *etc. The output signals* `Q` *and* `QN` *are replaced in the descriptions by bit-selections of the register, e.g.,* `PC_7_port` *and* `net27` *are replaced by* `PC[7]` *and* `not(PC[7])`*.*

```
PC_reg_7_label :  FD1M port map(CP=>CLK, D=>n569, Q=>PC_7_port, QN=>net27);
PC_reg_6_label :  FD1M port map(CP=>CLK, D=>n570, Q=>PC_6_port, QN=> net28);
....
PC_reg_1_label :  FD1M port map(CP=>CLK, D=>n578, Q=>PC_1_port, QN=>n496);
PC_reg_0_label :  FD1M port map(CP=>CLK, D=>n579, Q=>PC_0_port, QN=>n495);
```

becomes

```
PC <- term(n569) & term(n570) & ...  term(n578) & term(n579);
```

Fig. 9.5: Concatenation of register bits by the *SYN2IDS* translator

Memories are not synthesized by the Synopsys®-tool in our experiments. All memory-operations are replaced by assignments to interfaces before synthesis instead. The interfaces used for memory operations are

- the address-ports,
- the IN-, OUT-, or INOUT-data ports, and
- the write-enables.

Replacing `read`-operations by those interfaces before synthesis can be complex *only* for *behavioral* descriptions since the address has to be assigned *before* the value read is used.

Although memories are not synthesized, equivalence of memory operations on the memory-ports is verified according to [RHE99], too. Verifying an implementation against a specification with distinct order of memory operations as described in section 5.9 is possible at gate-level, too. The user has to declare in our prototype-version the memory-ports before translation (i.e., which signals are the address lines etc.) and the *SYN2IDS* translator generates the corresponding `read`- or `store`-operations for verification.

Translation is automatic, only the memory ports have to be denoted by the user.

9.5 Examples for Annotations to Generate Finite Sequences

Two examples for annotations of a description in *LLS* to generate the acyclic finite sequences for symbolic simulation as described in section 4.1.3 are given in the following.

Microprogram-architecture example

Fig. 9.6 and 9.7 demonstrate how the user can indicate the completion of an instruction in the implementation of Example 4.2 (section 4.1.3). The same annotations are necessary for the verification of the second example in section 7.2.2. Equivalence of a structural description of an architecture with microprogram control and the corresponding behavioral specification is checked in this example. No cycle equivalence is given. Therefore, the sequences to be compared are the complete executions of one instruction.

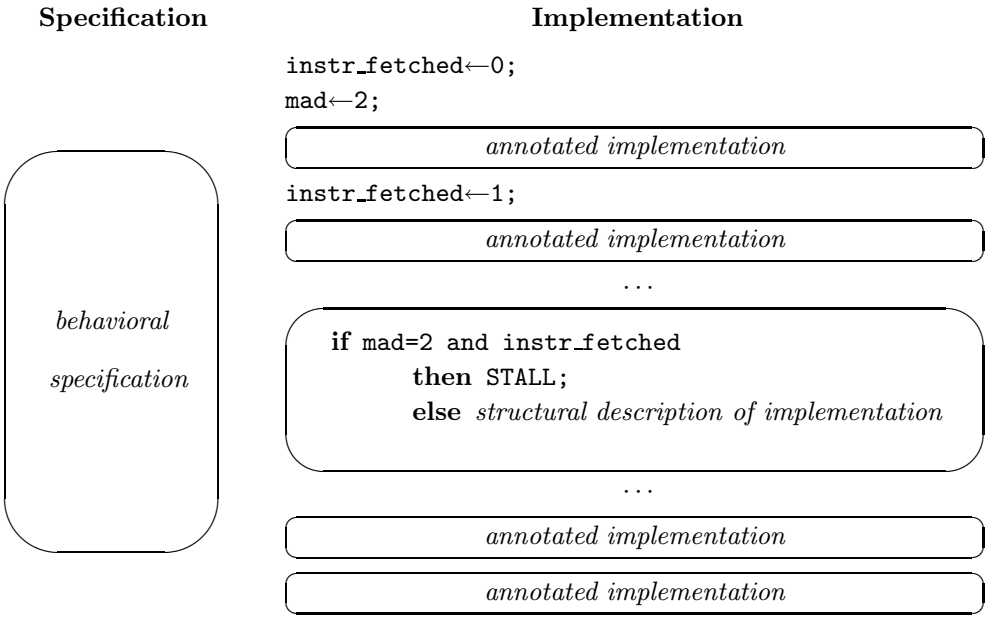


Fig. 9.6: Sequences to be compared for microprogram example

The execution of an instruction in the implementation of this microprogram-architecture takes depending on the instruction 8 to 10 cycles. Therefore, the description of the implementation is replicated according to the maximum number 10 times. The completion of an instruction has to be defined previously by an annotation. Only the annotation of one replicate is shown in the right-hand side of Fig. 9.6, the other copies (*annotated implementation*) are identical.[4] Ini-

[4]Many different annotations are possible to achieve the same result as in Fig. 9.6. For example, no annotation is required in the first cycle of the implementation. However, Fig. 9.7

tially, `instr_fetched` is cleared. Each instruction starts with the microprogram counter `mad=2` which is reached again after terminating the previous instruction. `instr_fetched` is set after fetching the first instruction. The *if-then-else*-clause evaluating `instr_fetched` prevents fetching an additional instruction if the first instruction takes less than 10 cycles, i.e., `mad=2` is reached again. The `then`-branch with the `STALL` is taken in this case, i.e., the register values are not changed in the remaining cycles. A replication of the behavioral specification is not necessary since it comprises one complete instruction.

Fig. 9.7 describes the annotations added to the *LLS*-description of the implementation. The design is described in the segment body of `La`. The sequence to simulate is given in the first two lines on the right-hand side. The segment `La` is used 10 times since this is the maximum number of cycles for the execution of an instruction. The auxiliary register `instr_fetched` is introduced to consider that some instructions take less than 10 cycles. It is cleared/set in `L_init`/`L_mark` to indicate whether an instruction has been started or not.

Implementation before annotations

```
La:     structural description
```

Implementation after annotations

Segments to simulate:

```
    L_init, La, L_mark, La, La, La,
    La, La, La, La, La, La
L_init: instr_fetched←0;
        mad←2;
L_mark: instr_fetched←1;
La:     if (mad=2) and instr_fetched
        then STALL;
        else structural description
```

Fig. 9.7: Annotations to generate the sequence to be simulated

DLX-Example

Section 7.2.1 gives experimental results for the verification of a structural DLX-description designed at Darmstadt University of Technology against a description of the DLX-instruction set. Section 4.1.3 describes how to generate for pipelined systems *in general* the two finite sequences to be compared according to the approach of [BD94]. The annotations required for symbolic simulation of the given DLX-example are discussed in the following.

The specification consists of flushing the pipeline followed by one serial execution. The implementation comprises fetching an instruction in the inner body of the pipeline loop followed by flushing the pipeline. Flushing the structural processor description is not automatic as for the behavioral descriptions presented in section 7.1 since the different states of the pipeline are not described separately.

demonstrates that replicating the same annotation is simpler for the user.

Only one structural description is given which subsumes all pipeline states. The number of cycles to simulate symbolically for flushing depends on possible stalls.

9 false negatives occurred due to incorrect flushing. These errors are more or less hard to consider in advance, but the equivalence checker identified the non-considered cases and correcting the flushing was simple. Note that the designer needed no insight in the verification process but only in his own design.

The improvements led to the flushing scheme sketched below. 4 cycles are required to flush a 5-stage pipeline *without* stalls.

Example 9.4
Fig. 9.8 shows one of the cases with two load-interlocks, where flushing takes more than 4 cycles.

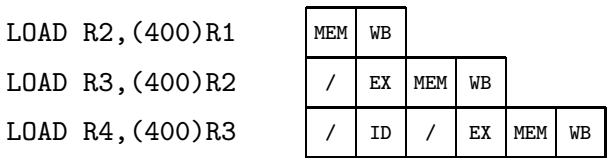


Fig. 9.8: Flushing with load-interlocks

Flushing can take up to 7 cycles. Therefore, generating the specification consists of linking the following segments:

- setting the stall-register and clearing the branch-flag if no branch is in the EX-stage, see below;
- 7 times the structural pipelined description, and
- the sequential (behavioral) description of the instruction set.

The branch-flag is set iff a branch terminating the ID-stage is taken, i.e., it can only be set if the operation in the EX-stage is a branch. Otherwise an impossible initial state is assumed, which leads to a false negative. Note that the necessity of this additional annotation was detected automatically, i.e., the designer got the hint by the false negative.

One instruction is fetched before flushing in the implementation. But this instruction needs not be fetched in the first cycle. There might be a stall due to a load interlock or a taken branch, which delays the instruction fetch. Therefore, the worst case number of cycles to simulate is 9.

Example 9.5
Fig. 9.9 gives an example, where fetching one instruction and flushing afterwards takes 9 cycles. The branch is taken.

The cycle has to be determined, when the instruction is fetched and flushing has to begin. No instruction is fetched during a load-interlock. Furthermore, an

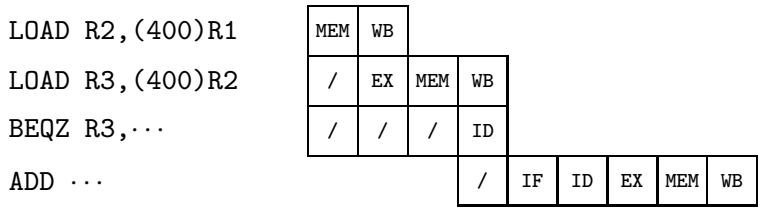


Fig. 9.9: Worst case number of cycles for fetching one instruction and flushing

instruction fetched is not executed after a taken branch. Therefore, an annotation is required each time after the first cycles, which sets the stall-register only if no taken branch or jump is in the EX-stage and no load-interlock occurred. An instruction fetched is not squeezed at least after three cycles. Flushing can begin at the latest after 5 cycles. The implementation consists of linking:

- clearing the branch-flag if no branch is in the EX-stage;
- 5 times
  - the structural pipelined description followed by
  - an annotation setting the stall-input if there was no taken branch, jump, or load-interlock;[5]
- 4 times the structural pipelined description.

[5]It is not necessary to test all these conditions in each of the 5 cycles. Therefore, the actual implementation of the flushing is slightly simpler.

9.6 Interpreted Functions

Table 9.1 summarizes the functions interpreted currently by the symbolic simulator. Functions defined in *LLS*, but not listed in Tab. 9.1 are considered as uninterpreted functions. The same holds for user-defined functions.[6] A detailed description of the functions defined in *LLS* is given in [Hin98b].

A selection of *uninterpreted* functions is marked during pre-processing. The second approach for equivalence detection described in section 5.1.2 is applied to those terms.

The symbolic simulator does not distinguish between registers of type bit-vector or integer. However, some equivalence detection techniques, particularly those based on decision diagrams, cannot be used on integers. A solution is to provide the information about the maximum value of an integer typed register in the USE-declaration of the *LLS*-description. Every argument or result of type vector (v) in Tab. 9.1 can be either a bit-vector or an integer with range information. A type mismatch can be resolved in *LLS* by using the functions `BITINT` or `INTBIT`. These two functions are removed during pre-processing. Therefore, the functions in Tab 9.1 can have results with different types. Note that compatibility of types is checked in the original description (particularly concerning the bit-vector length) by the *LLS-to-IDS*-compiler and by the *FDS-to-EDS*-translator, see also [Hin98b].

The main differences between the functions in Tab. 9.1 and the corresponding *LLS* functions in addition to typing are:

- Boolean functions can have only two arguments in *LLS*. Successive applications are transformed during pre-processing to allow a faster symbolic simulation, e.g., `(and a (and b c))` becomes `(and a b c)`;
- array-selections in *LLS* are transformed to `read`- or `store`-operations in the internal data structure of the symbolic simulator. The same holds for element/slice selections if an index is not a number. An exception are the two-dimensional concatenations used by the `mpx2`-function. *LLS* permits to select not only an entire word, but also single bits from an array; an element/slice selection is added in this case, e.g., `mem[adr,5]` becomes `(read mem adr)[5]`;
- the two-dimensional concatenation is only used for the `mpx2`-function.

[6] With exception of the functions `unknown` (section 5.8) and `violate` (section 7.4), which are not defined in *LLS*.

Abbreviations in Tab. 9.1

n	only a number permitted
b	Boolean
v	bit-vector or integer with range information
i	integer
vi	bit-vector or integer
2dimv	two-dimensional vector of integers/bit-vectors (produced by concatenation)
mem	memory

Function	Arguments	Result	Example
concatenation	vb,vb	v	011B3#101B3≅011101B6
two-dimensional	vib,···,vib	2dimv	011B3##101B3
element selection	vi,n	b	(0010B4)[3]≅0B1
slice selection	vi,n,n	v	(0010B4)[1:0]≅10B2
array selection `read`	vi,mem	vbi	see section 4.1.5
array selection `store`	vi,mem,vbi	mem	see section 4.1.5
addition, carry in/out	vb,vb,b	v	adc(111B3,001B3,1B1)≅1001B4
addition modulo	vb,vb	vb	addmod(011B3,101B3)≅000B3
subtraction, carry in/out	vb,vb,b	v	sbb(101B3,101B3,1B1)≅1111B4
subtraction modulo	vb,vb	vb	submod(101B3,110B3)≅111B3
incrementation-with-carry	vb	v	inc(111B3)≅1000B4
incrementation modulo	vb	vb	incmod(100B3)≅101B3
decrementation modulo	vb	vb	decmod(100B3)≅011B3
plus	i,i	i	4+3≅7
minus	i,i	i	4-3≅1
multiplication	vb,vb	v	010B3∗011B3≅000110B6
right shift	b,vb	vb	rsh(1B1,011B3)≅101B3
left shift	vb,b	vb	lsh(011B3,0B1)≅110B3
rotate left	vb	vb	rol(011B3)≅110B3
rotate right	vb	vb	ror(011B3)≅101B3
multiplexer	v,vb	b	mpx1(0010B4,11B2)≅0B1
two-dimensional	2dimv,vib	vib	mpx2(001B3##100B3,1B1)≅100B3
=	vib,vib	b	(101B3=011B3)≅0B1
≠	vib,vib	b	(101B3≠011B3)≅1B1
>	vib,vib	b	(101B3>011B3)≅1B1
<	vib,vib	b	(101B3<011B3)≅0B1
≥	vib,vib	b	(101B3≥011B3)≅1B1
≤	vib,vib	b	(101B3≤011B3)≅0B1
Boolean and	b,···,b	b	1B1&0B1&1B1 ≅ 0B1
Boolean or	b,···,b	b	1B1|0B1|1B1 ≅ 1B1
Boolean exor	b,···,b	b	1B1⊕0B1⊕1B1 ≅ 0B1
Boolean negation	b	b	∼1B1 ≅ 0B1
Boolean and on vectors	v,···,v	v	101B3&001B3 ≅ 001B3
Boolean or on vectors	v,···,v	v	101B3|001B3 ≅ 101B3
Boolean exor on vectors	v,···,v	v	101B3⊕001B3 ≅ 100B3
Boolean neg. on a vector	v	v	∼101B3 ≅ 010B3
violate	vib,vib	b	see section 7.4
unknown	vib	vib	unknown(42)

Tab. 9.1: Types of functions. Examples partly taken from [ES92]

9.7 Properties of *EqvClasses*, *CondBits*, *RegVals*, and *Term Representatives*

Tab. 9.2 to Tab. 9.5 summarize the most important properties of *RegVals*, *Term Representatives*, *EqvClasses*, and *CondBits* during symbolic simulation.

Property	**Description**
WORD-CONN-WITH	term assigned on current path
EQC	*EqvClass* of *RegVal*
LENGTH	number of bits of register
NR	index of *RegVal*; 0 for initial *RegVal*
0,1,2,3,...	terms, which are *bit-selections* of the *RegVal*; the number corresponds to the number of the bit; example: the property `3` of term `reg` is the *Term Representative* of `reg[3]`, see also section 5.7;
ORG-REG	corresponding initial *RegVal*
PRIMED-SPEC PRIMED-IMPL	last *RegVal* of this register in specification or implementation (only marked at initial *RegVals*)
STORES-SPEC STORES-IMPL	`store`-operations to this memory (*RegVal*) in specification/ implementation (only marked at initial *RegVals*)
READ-SPEC READ-IMPL	`read`-operations from this memory (*RegVal*) in specification/ implementation (only marked at initial *RegVals*)

Tab. 9.2: Properties of *RegVals*

Property	**Description**
EQC	*EqvClass* of term
TERM-ALREADY-FOUND	flag indicating if term has been already found on current path
LENGTH	number of bits of term
CONST-IN-ARITH-EXPR	see section 5.3
POS-ARGS-IN-ARITH-EXPR	see section 5.3
NEG-ARGS-IN-ARITH-EXPR	see section 5.3
POS-SYM-BIT-IS	*positive-bit-equivalent*, see section 5.2
NEG-SYM-BIT-IS	*negative-bit-equivalent*, see section 5.2
0,1,2,3,..	see corresponding entry in Tab. 9.2
ASSOC	terms are replaced for technical reasons by an arbitrary chosen distinct variable called *Term Representative*, see appendix 9.2; the property ASSOC of these variables gives the corresponding expression of the term

Tab. 9.3: Properties of terms (*Term Representatives*)

Property	Description
MEMBERS	members of the *EqvClass*; can be *RegVals* or *Term Representatives*
CONSTANT	constant of the *EqvClass*; NIL if terms in *EqvClass* are not equivalent to a constant
VALUE-BOUNDS	restrictions of the range of the terms in the *EqvClass*, see section 5.5
INEQU	list of inequivalent *EqvClasses*; inequivalences between *EqvClasses* with constants need not be considered, see section 4.3
DEP-READ	`read`-operations, which use one of the *RegVals*/terms of the *EqvClass* as address, see section 5.9.2
CAT1-CONST-PARTS	connected areas of bits of the *RegVals*/terms in the *EqvClass*, which are equivalent to constants, see section 5.6

Tab. 9.4: Properties of *EqvClasses*

Property	Description
VALUE	value of *CondBit*: *undefined*, *true*, or *false*
COND	condition associated with the *CondBit* • a *RegVal* (length one bit), • a *Term Representative* (length one bit), or • comparison of two *Term Representatives* or *RegVals*

Tab. 9.5: Properties of *CondBits*

9.8 Verification Approach of Burch/Dill for Systems with Pipelining

Fig. 9.10 demonstrates the verification of a system with pipelining by the approach of [BD94]. An old implementation state is transformed in two manners into a new specification state. F_{impl} and F_{spec} describe the transition functions, and I_{stall}/I are arbitrary input combinations stalling/not stalling the processor. Section 4.1.3 describes the verification of a system with pipelining by comparing two finite sequences obtained by flushing. Comparing the two new specification states of Fig. 9.10 is basically the same, see for more details section 4.1.3.

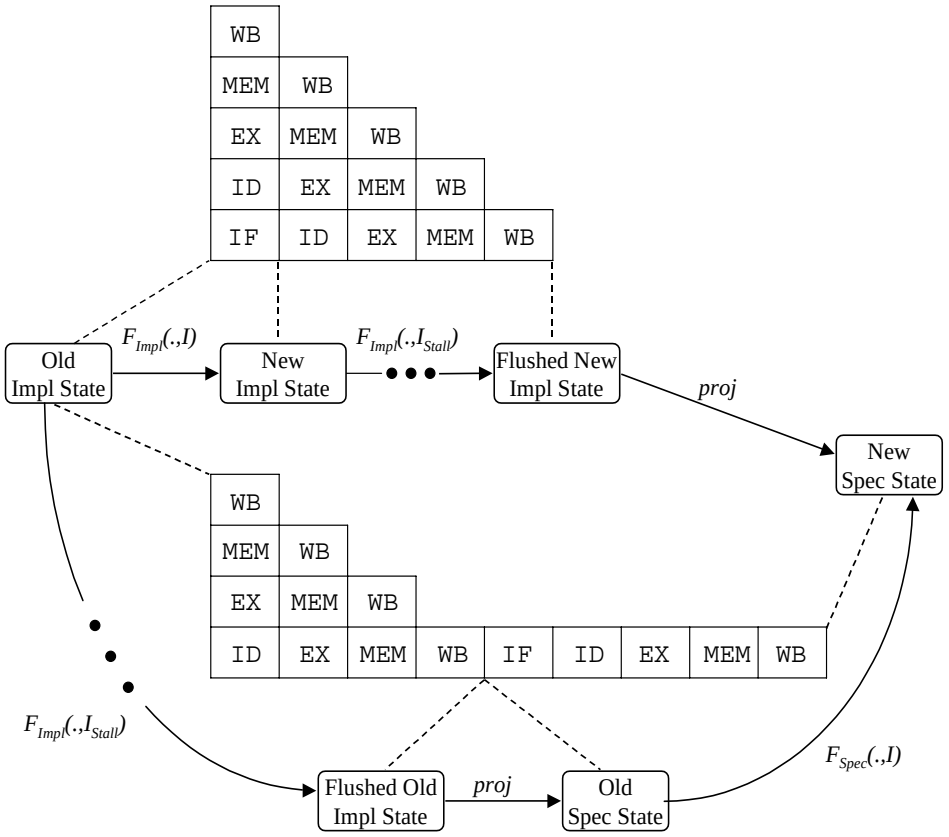


Fig. 9.10: Illustration of verification of systems with pipelining by [BD94]

9.9 Verification of the MPA example

Fig. 9.11 summarizes the transformation steps for the verification of the MPA example in section 7.3. The results of Tab. 7.4 refer to the equivalence checks indicated by the two bold arrows in Fig. 9.11.

9.10 Rejected or Improved Implementation Details

The following list describes implementation details which have been either tested and rejected, or which have been improved during the development of the symbolic simulator:

- initially, the general procedure for unifying two *EqvClasses* was applied also if the union operation was due to an assignment. Practically, this union operation is significantly simpler because the *EqvClass* of the *RegVal* on the left-hand side of the assignment is guaranteed to be not modified previously, see section 4.3;

- single-*bit-selections* are considered as functions with only one argument for equivalence detection. The second argument, i.e., the number of the

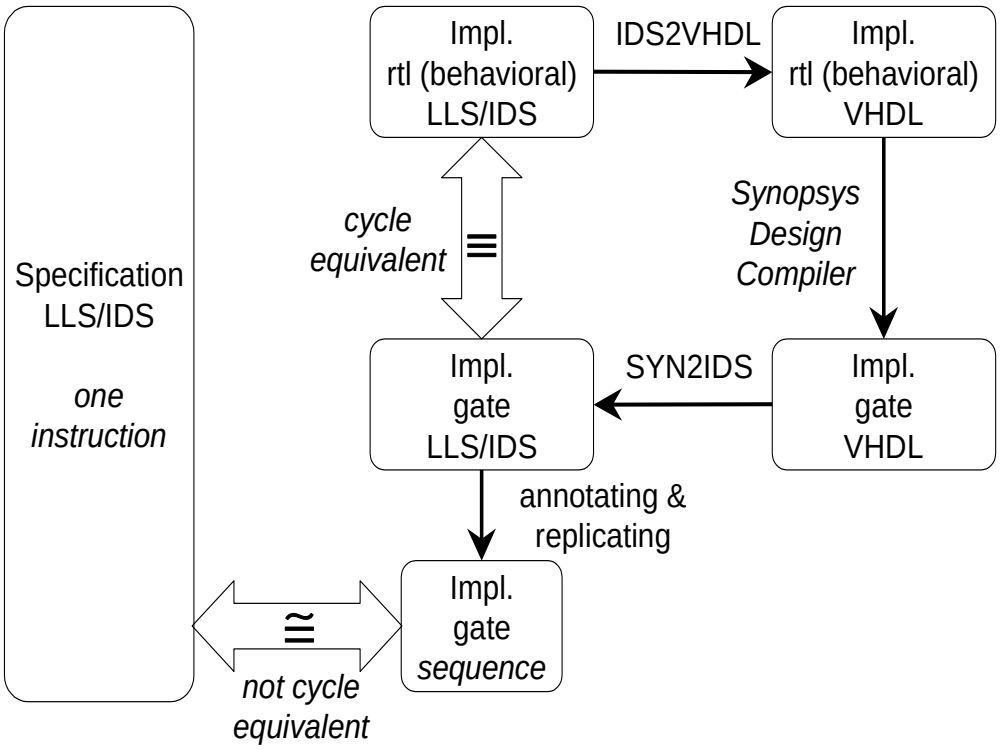


Fig. 9.11: Verification of MPA example

bit to select is a constant and considered in the function symbol, e.g., `(bit-selection-4 ir)` instead of `(bit-selection ir 4)`. This permits a faster equivalence detection as described in section 5.7;

- applying the general equivalence detection techniques (section 5.1) to multiplexers is not efficient. A single special *if-then-else*-clause is used instead to force a decision about the value of the control bits, see section 5.4;
- initially, it was controlled after each case-split whether a term with domain 2^n has been set inequivalent to $2^n - 1$ constants. The term is equivalent to the remaining constant in this case. A more efficient procedure is described in section 5.10;
- the special function `unknown` was introduced to avoid unnecessary applications of the general equivalence detection techniques for unspecified parts, see section 5.8;
- all constants described as bit-vectors in *LLS* are translated to integers during pre-processing (e.g., `(CONST 1 1 0)` becomes `6`) to permit a faster comparison of constants and to reduce the size of the descriptions to simulate, see appendix 9.3;
- the procedure for context saving and alternatives rejected after testing are described in [Smi98].

Bibliography

[ABRM98] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama. Verification of RTL generated from scheduled behavior in a high-level synsthesis flow. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1998.

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1954.

[AGM96] P. Ashar, A. Gupta, and S. Malik. Using complete-1-distinguishability for FSM equivalence checking. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1996.

[AJK+00] M. D. Aagaard, R. B. Jones, R. Kaivola, K. R. Kohatsu, and C.-J. H. Seger. Formal verification of iterative algorithms in microprocessors. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 2000.

[AJM+00] M. D. Aagaard, R. B. Jones, T. F. Melham, J. W. O'Leary, and C.-J. H. Seger. A methodology for large-scale hardware verification. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*. Springer Verlag, 2000.

[AJS98] M. D. Aagaard, R. B. Jones, and C-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1998.

[AJS99] M. D. Aagaard, R. B. Jones, and C-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999.

[BB94] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1994.

[BBB+87] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1987.

[BBS91] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1991.

[BC94] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CMU-CS-94-160, Carnegie Mellon University, 1994.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1995.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. Originally presented at the 1990 Symposium on Logic in Computer Science (LICS'90).

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1990.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. Computer Aided Verification (CAV)*, volume 818 of *LNCS*. Springer Verlag, 1994.

[BDL96] C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*. Springer Verlag, 1996.

[BDL98] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1998.

[BDQ99] V. Bertacco, M. Damiani, and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999.

[Ber91] R. A. Bergamaschi. The effects of false paths in high-level synthesis. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1991.

[BF89] S. Bose and A. L. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *Proc. International Conference on Computer Design (ICCD)*, 1989.

[BGV99] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc. Computer Aided Verification (CAV)*, volume 1633 of *LNCS*. Springer Verlag, 1999.

[BHK94] B. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic Inc., 1994.

[BKM96] B. Brock, M. Kaufmann, and J. S. Moore. ACL2 theorems about commercial microprocessors. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*. Springer Verlag, 1996.

[Bla00] C. Blank. Formal verification of register binding. In *Proc. Workshop on Advances in Verification (Wave'2000), Chicago*, 2000.

[BM75] R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1):129–144, 1975.

[BM79] R. S. Boyer and J. S. Moore. *A computational logic*. Academic Press, New York, 1979.

[BM97] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press, London, second edition, 1997.

[Bow00] J. Bowen. Formal methods.
URL: http://archive.comlab.ox.ac.uk/formal-methods.html.
Centre for Applied Formal Methods, SCISM, South Bank University, London, 2000.

[BRHE00] C. Blank, G. Ritter, H. Hinrichsen, and H. Eveking. Formale Verifikation der Register-Allokation. In *Proc. ITG/GI/GMM-Workshop, Frankfurt*, 2000.

[Bry85] R. E. Bryant. Symbolic verification of MOS circuits. In *Proc. Chapel Hill Conference on VLSI*, pages 419–438. Computer Science Press, 1985.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bry90a] R. E. Bryant. Symbolic simulation - techniques and applications. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1990.

[Bry90b] R. E. Bryant. Verification of synchronous circuits by symbolic logic simulation. In *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pages 14–24. Springer-Verlag, 1990.

[Bur96] J. R. Burch. Techniques for verifying superscalar microprocessors. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1996.

[CBM89a] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*. North-Holland, 1989.

[CBM89b] O. Coudert, C. Berthet, and J.-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer Verlag, 1989.

[CBM90] O. Coudert, C. Berthet, and J. C. Madre. Formal boolean manipulations for the verification of sequential machines. In *Proc. European Design Automation Conference (EDAC)*, 1990.

[CCPQ99] G. Cabodi, P. Camurati, C. Passerone, and S. Quer. Computing timed transition relations for sequential cycle-based simulation. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, 1999.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CJB79] W. C. Carter, W. H. Joyner Jr., and D. Brand. Symbolic simulation for correct machine design. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1979.

[CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *IEEE International Conference on Automated Deduction (CADE)*, volume 1104 of *LNAI*. Springer Verlag, 1996.

[CMR97] D. Cyrluk, O. Möller, and H. Rueß. An efficient decision procedure for the theory of fixed-size bit-vectors. In *Proc. Computer Aided Verification (CAV)*, volume 1254 of *LNCS*. Springer Verlag, 1997.

[Cor81] W. E. Cory. Symbolic simulation for functional verification with ADLIB and SDL. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1981.

[Cor92] Digital Equipment Corporation. *Alpha architecture handbook*, 1992.

[CRS98] F. Corno, M. S. Reorda, and G. Squillero. VEGA: a verification tool based on genetic algorithms. In *Proc. International Conference on Computer Design (ICCD)*, 1998.

[CRS99] F. Corno, M. S. Reorda, and G. Squillero. Approximate equivalence verification of sequential circuits via genetic algorithms. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, 1999.

[CW96] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), 1996.

[Dar79] J. A. Darringer. The application of program verification techniques to hardware verification. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1979.

[DK78] J. A. Darringer and J. C. King. Applications of symbolic execution to program testing. *IEEE Computer*, 11(4):51–60, 1978.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, volume 85 of *LNCS*. Springer Verlag, 1980.

[EHR98] H. Eveking, H. Hinrichsen, and G. Ritter. Formally correct construction of pipelined processors. Technical Report 98-6-1, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 1998.

[EHR99] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, 1999.

[ES92] H. Eveking and U. Schellin. The SMAX internal data structure. Technical Report THD-2.B.2.b-04, Darmstadt University of Technology, 1992.

[Eve91] H. Eveking. *Verifikation digitaler Systeme*. B.G. Teubner Stuttgart, 1991.

[GAK99] M. K. Ganai, A. Aziz, and A. Kuehlmann. Enhancing simulation with BDDs and ATPG. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[GMA97] A. Gupta, S. Malik, and P. Ashar. Toward formalizing a validation methodology using simulation coverage. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1997.

[Gre98] D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *LNCS*. Springer Verlag, 1998.

[HER99] H. Hinrichsen, H. Eveking, and G. Ritter. Formal synthesis for pipeline design. In *Proc. DMTCS+CATS'99, Auckland*, volume 21, number 3 of *Australian Computer Science Communications*, pages 247–261. Springer Verlag, 1999.

[Hin98a] H. Hinrichsen. Formally correct construction of a pipelined DLX architecture. Technical Report 98-5-1, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 1998.

[Hin98b] H. Hinrichsen. Language of Labelled Segments documentation, URL: http://www.rs.e-technik.tu-darmstadt.de/~hinni/document/index.html. Technical report, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 1998.

[Hin00] H. Hinrichsen. *Ein transformativer Ansatz für die Synthese und Verifikation algorithmischer Hardwarebeschreibungen*. PhD thesis, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 2000.

[HK76] S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, 1976.

[Hör97] S. Höreth. Implementation of a multiple-domain decision diagram package. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, 1997.

[Hör98] S. Höreth. Hybrid graph manipulation package demo. http://www.rs.e-technik.tu-darmstadt.de/~sth/demo.html, Darmstadt, 1998.

[Hör99] S. Höreth. *Effiziente Konstruktion und Manipulation von binären Entscheidungsgraphen*. PhD thesis, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 1999.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufman, CA, second edition, 1996.

[HRE99] H. Hinrichsen, G. Ritter, and H. Eveking. Automatische Synthese und Verifikation von RISC-Prozessoren. In *Proc. GI/ITG/GMM Workshop, Braunschweig*, 1999.

[HRE00] H. Hinrichsen, G. Ritter, and H. Eveking. False-path elimination and simplification of sequential acyclic descriptions with complex branching logic. In *Proc. Workshop on Algorithm Architecture Adequation (AAA) 2000, Rocquencourt, France*, 2000.

[HS97] S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation. In *Formal Hardware Verification. Methods and Systems in Comparison*, volume 1287 of *LNCS*. Springer Verlag, 1997.

[HSG98] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In *Proc. Computer Aided Verification (CAV)*, volume 1427 of *LNCS*. Springer Verlag, 1998.

[HSG99] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *Proc. Computer Aided Verification (CAV)*, volume 1633 of *LNCS*. Springer Verlag, 1999.

[ID96] C. N. Ip and D. L. Dill. State reduction using reversible rules. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1996.

[Inc93] Microchip Technology Inc. *Microchip data book*, 1993.

[JDB95] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1995.

[JG92] P. Jain and G. Gopalakrishnan. Some techniques for efficient symbolic simulation-based verification. In *Proc. International Conference on Computer Design (ICCD)*, 1992.

[JSD98] R. B. Jones, J. U. Skakkebæk, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *LNCS*. Springer Verlag, 1998.

[KG99] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2), 1999.

[Kin75] J. C. King. A new approach to program testing. *SIGPLAN Notices. Proc. International Conference on Reliable Software*, 10(6):228–233, 1975.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KM97] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[Lev00] J. Levihn. Übersetzer für C in eine Beschreibungssprache für erweiterte Zustandsdiagramme. Master's thesis, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 2000.

[LO96] J. Levitt and K. Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1996.

[LO97] J. Levitt and K. Olukotun. Verifying correct pipeline implementation for microprocessors. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1997.

[Moo98] J. S. Moore. Symbolic simulation: an ACL2 approach. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *LNCS*. Springer Verlag, 1998.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *IEEE International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*. Springer Verlag, 1992.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. v. Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[OZGS99] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compilance of floating-point hardware. *Intel Technology Journal*, First Quarter, 1999.

[PB99] M. Pandey and R. E. Bryant. Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):918–935, 1999. See also (same authors/title) Proc. Computer Aided Verification (CAV), volume 1254 of LNCS. Springer Verlag, 1997.

[PRBA97] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1997.

[PRBB96] M. Pandey, R. Raimi, D. L. Beatty, and R. E. Bryant. Formal verification of PowerPC^{TM} arrays using symbolic trajectory evaluation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1996.

[REH99] G. Ritter, H. Eveking, and H. Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *LNCS*. Springer Verlag, 1999.

[RHE99] G. Ritter, H. Hinrichsen, and H. Eveking. Formal verification of descriptions with distinct order of memory operations. In *Proc. ASIAN'99*, volume 1742 of *LNCS*. Springer Verlag, 1999.

[Rit00] G. Ritter. Sequential equivalence checking by symbolic simulation. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*. Springer Verlag, 2000.

[RJ95] M. Rahmouni and A. A. Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In *Proc. European Design Automation Conference (Euro-DAC)*, 1995.

[SB95] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[SD98] U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the Mur φ verifier. In *Proc. Computer Aided Verification (CAV)*, volume 1427 of *LNCS*. Springer Verlag, 1998.

[Sho79] R.E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, 1979.

[Sho84] R.E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[SJD98] J. U. Skakkebæk, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *Proc. Computer Aided Verification (CAV)*, volume 1427 of *LNCS*. Springer Verlag, 1998.

[SM95a] M. Srivas and S. P. Miller. Applying formal verification to a commercial microprocessor. In *IFIP International Conference on Computer Hardware Description Languages, Chiba, Japan*, August 1995.

[SM95b] M. Srivas and S. P. Miller. Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods. In *Workshop on industrial-strength formal specification techniques (WIFT)*, pages 2–16. IEEE Computer Society, 1995.

[Smi98] K. Smith. Optimierung eines Verfahrens zur formalen Äquivalenzprüfung von Prozessorbeschreibungen. Master's thesis, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 1998.

[VB98] M. N. Velev and R. E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *LNCS*. Springer Verlag, 1998.

[VB99a] M. N. Velev and R. E. Bryant. Exploiting postive equality and partial non-consistency in the formal verification of pipelined microprocessors. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999.

[VB99b] M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *LNCS*. Springer Verlag, 1999.

[VB00] M. N. Velev and R. E. Bryant. Formal verification of superscalar processors with multicycle functional units, exceptions, and branch prediction. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 2000.

[WAK98] L.-C. Wang, M. S. Abadir, and N. Krishnamurthy. Automatic generation of assertions for formal verification of PowerPC^{TM} microprocessor arrays using symbolic trajectory evaluation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1998.

[WB96] P. J. Windley and J. R. Burch. Mechanically checking a lemma used in an automatic verification tool. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*. Springer Verlag, 1996.

[WDB00] C. Wilson, D. L. Dill, and R. E. Bryant. Symbolic simulation with approximate values. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*. Springer Verlag, 2000.

Publications

Computer Engineering

[Rit00] G. Ritter. Sequential equivalence checking by symbolic simulation. In *Proc. Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*. Springer Verlag, 2000.

[REH99] G. Ritter, H. Eveking, and H. Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *LNCS*. Springer Verlag, 1999.

[RHE99] G. Ritter, H. Hinrichsen, and H. Eveking. Formal verification of descriptions with distinct order of memory operations. In *Proc. ASIAN'99*, volume 1742 of *LNCS*. Springer Verlag, 1999.

[RHE99b] G. Ritter, H. Hinrichsen, and H. Eveking. Formale Verifikation automatisch generierter Pipelinesysteme durch symbolische Simulation. In *Proc. 9. Entwurf Integrierter Schaltungen (EIS) Workshop. Darmstadt, September 22–24*, 1999.

[EHR99] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, 1999.

[HER99] H. Hinrichsen, H. Eveking, and G. Ritter. Formal synthesis for pipeline design. In *Proc. DMTCS+CATS'99, Auckland*, volume 21, number 3 of *Australian Computer Science Communications*, pages 247–261. Springer Verlag, 1999.

[BRHE00] C. Blank, G. Ritter, H. Hinrichsen, and H. Eveking. Formale Verifikation der Register-Allokation. In *Proc. ITG/GI/GMM-Workshop, Frankfurt*, 2000.

[HRE00] H. Hinrichsen, G. Ritter, and H. Eveking. False-path elimination and simplification of sequential acyclic descriptions with complex branching logic. In *Proc. Workshop on Algorithm Architecture Adequation (AAA) 2000, Rocquencourt, France*, 2000.

[Rit00b] G. Ritter. Vérification formelle dans la synthèse automatique des systèmes avec pipeline. In *Proc. JNRDM-Workshop 2000, Montpellier, May 4–5*, 2000.

[HRE99] H. Hinrichsen, G. Ritter, and H. Eveking. Automatische Synthese und Verifikation von RISC-Prozessoren. In *Proc. GI/ITG/GMM Workshop, Braunschweig*, 1999.

Technical Reports

[Rit99] G. Ritter. Functional description and macro architecture of an industrial viterbi decoder (20 pp.). Technical report, TIMA laboratory, Grenoble, France, 1999.

[EHR98] H. Eveking, H. Hinrichsen, and G. Ritter. Formally correct construction of pipelined processors. Technical Report 98-6-1, Darmstadt University of Technology, Dept. of Electrical and Computer Engineering, 1998.

Business Management

[HR97] M. Hupe and G. Ritter. Der Einsatz risikoadjustierter Kalkulationszinsfüße bei Investitionsentscheidungen. *Betriebswirtschaftliche Forschung und Praxis BFuP (journal)*, 49(5):593–612, 1997.

Abbreviations

$\cong_C$	see description on page 13
$\ncong_C$	see description on page 14
$\equiv_C$	see Definition 2.6 on page 13
$\not\equiv_C$	see Definition 2.7 on page 14
**BMD*	multiplicative binary moment diagram
bit-selection	selection of bits of a term, for example, `a[16:8]` or `a(16 downto 8)` in VHDL-notation
CondBit	*Condition-Bit*, represents a boolean term which is used in conditions; value can be *true*, *false*, or *undefined*, see section 4.4
condition term	a propositional connective (`not`, `nand`, `nor`, `and`, `or`, `xor`) applied to a list of *CondBits* and/or other *condition terms*, see section 4.4
ctrl-one-bit	description see section 5.10
ctrl-zero-bit	description see section 5.10
dd-check	equivalence detection techniques using *OBDD*-vectors, see chapter 6
dd-cutpoint	cutpoint used to simplify a *dd-check*, see section 6.2
EDS	*equchecker description structure*, input format to the symbolic simulator, see section 4.1.2
equivalent	see $\cong_C$
EqvClass	equivalence class, see section 2.6
IDS	*intermediate data structure*/format, see section 4.1.2
inequivalent	see $\ncong_C$
LLS	*language of labelled segments*, the input description language, see section 4.1.1
negative-bit-equivalent	equivalence information of a bit; used to detect equivalences of Boolean terms and concatenations, see section 5.2 and 5.6

OBDD	ordered binary decision diagram
positive-bit-equivalent	see *negative-bit-equivalent*
read access	relevant memory state for a `read`-operation, see section 5.9.2
RegVal	register value; different symbolic register values are introduced for the initial register value and after each assignment to a register, see section 4.1.4
simulation-cutpoint	representative for a sub-expression, which occurs multiple times in other expressions; used to avoid repeated evaluation of the sub-expression, see appendix 9.3
STE	Symbolic Trajectory Evaluation, see section 3.2
SVC	Stanford Validity Checker, see section 3.3
SYN2IDS translator	compiles a subset of the VHDL-output of the Synopsys® Design Compiler™ to *IDS*-format
Term Representative	arbitrary chosen distinct variable which represents a term; used for technical reasons, see appendix 9.2
TUDD-package	*OBDD*-package developed at Darmstadt University of Technology
valuebound	information about the range of a term; used to detect equivalences of comparisons, i.e., >, <, >=, and <=, see section 5.5

Curriculum Vitae

Name	Gerd RITTER
Date and Place of Birth	8th August 1969 in Frankfurt/Main
Nationality	German
Marital Status	Married, one child
Foreign Languages	English, French (fluently)

Academic Qualifications

Sep 1998 until present	Combined bi-national PhD with TIMA Laboratory, Université Joseph Fourier, France, and Darmstadt University of Technology, Germany, supported by Deutsch-Französisches Hochschulkolleg.
Dec 1995	Diploma in Business Administration with Electrical Engineering, Darmstadt University of Technology (1st of 85), Germany.
Sep 1989 - Dec 1995	Studies of Business Administration with Electrical Engineering, Darmstadt University of Technology.
Sep 1993 - Aug 1994	Studies at University of Bordeaux I, France, supported by the ERASMUS Program.
Sep 1992	Member of the "Studienstiftung des Deutschen Volkes" (Honour Association).

Employment History

Jan 1996 until present	Research and Teaching Assistant, Department of Electrical and Computer Engineering, Darmstadt University of Technology.
Sep - Nov 1995	Practical Course with Daimler-Benz AG (Holding, Division Trust Planning), Stuttgart, Germany.
Feb 1991 - Dec 1995	Student Assistant in the Field of General Management, Darmstadt University of Technology.

Military Service German Bundeswehr, July 1988 - September 1989.

Abstract

A new approach to sequential verification of designs at different levels of abstraction by symbolic simulation is proposed. The automatic formal verification tool has been used for equivalence checking of structural descriptions at rt-level and their corresponding behavioral specifications. Gate-level results of a commercial synthesis tool have been compared to specifications at behavioral or structural rt-level. The specification need not be synthesizable nor cycle equivalent to the implementation. In addition, a future application of the method to property verification is proposed.

Symbolic simulation is guided along logically consistent paths in the two descriptions to be compared. An open library of different equivalence detection techniques is used in order to find a good compromise between accuracy and speed. Decision diagram (OBDD) based techniques detect corner-cases of equivalence. Graph explosion is avoided by using the results of the other equivalence detection techniques and by representing only small parts of the verification problem by decision diagrams. The cooperation of all techniques as well as good debugging support are made feasible by notifying detected relationships at equivalence classes instead of manipulating symbolic terms.

Keywords:

formal verification, symbolic simulation, equivalence checking, sequential verification, hardware verification, gate-level, rt-level

Kurzfassung (german abstract) on page vi

Résumé (french abstract) on page vii

Research performed at

Dept. of Electrical and
Computer Engineering
Darmstadt University of Technology

TIMA Laboratory
Université Joseph Fourier
Grenoble

ISBN 2-913329-65-9